Invariants-based Architecture for Semantic Malware Resistance

Rachid Rebiha *Arnaldo V. Moura*

Technical Report - IC-15-02 - Relatório Técnico

January - 2015 - Janeiro

Invariants-based Architecture for Semantic Malware Resistance

Rachid Rebiha[*] Arnaldo Vieira Moura[†]

Abstract

In this work we provide *theoretical basis* for the design of static and dynamic platforms that can exhibit a suitable architecture for automatic in-depth malware analysis. We show how formal methods involving static and dynamic program analysis, decision procedures and automated verification techniques can be used to build such architectures. We present automatic formal specification, classification and detection methods for malware analysis. We provide several methods to generate *formal malware signatures* capable of capturing and modelizing the core of the malicious behavior in a sound and concise way. We propose to generate *malware invariants and abstract models* directly from the specified malware code in order to use them as *semantic aware* signatures. Importantly, these invariants and abstracted structures would remain unchanged in many *obfuscated* versions of the code. Then, we present formal model extraction methods in order to modelize the program being inspected. These computational models are basically abstract call graphs and pushdown systems annoted with invariants at each system call locations. Thus, for each specification methods and formal models introduced, we are able to reduce the problem of malware detection to classical problem very standard for the formal method research community. We also present a *host-based intrusion detection systems*, where system calls are preceded by the check of pre-computed invariants. We prove that any malware analysis or intrusion detection system will be strongly re-enforced by the presence of pre-computed invariants, and will also be weakened by their absence. The research security community will benefit from mathematical formalisms that could serve as a scientific basis for their classification. The other main purpose of this paper is to enlight the formal method research community about writing static malware analyzer using several formal techniques and combined logics.

1 Introduction

Invariant properties are assertions, expressed in a specified logic, that hold true on every possible run of a system. A *malware* is a program that has malicious intent. Examples of such programs include viruses, Trojans horses, and worms. Malicious intent to computers can be virulent threats to society. We deeply need to understand *malicious behaviors* in more detail.

[*]Instituto de Computação, Universidade Estadual de Campinas, 13081970 Campinas, SP. Pesquisa desenvolvida com suporte financeiro da FAPESP, processos 2011089471

[†]Instituto de Computação, Universidade Estadual de Campinas, 13081970 Campinas, SP.

All present security systems, like anti-virus and other detection systems, suffer from a lack of automation in their malware analysis. In order to provide automatic in-depth malware analysis and precise detection systems, one needs to be able to automatically extract the malicious behaviors, and not just its syntactic signature.

Current malware detectors are "*signature-based*": the presence of the malicious behavior is detected if the malicious code matches some byte-signature. Such current malware detectors are based on sound methods, with byte-signatures located in a database of regular expressions which specify byte or instructions sequences. But the main problem is that malware writers can then use *Obfuscation* [1] to evade current detectors. To evade detection, hackers frequently use obfuscation to morph malware. By so doing, they can evade detection by injecting code into malwares in a way that preserves malicious behavior and makes the previous signature irrelevant. Further, current intrusion detection systems are based on too coarse abstraction methods.

The number of malwares variants that use obfuscation increases exponentially each time a new malware type appear. Malware writers can easily generate new undetectable viruses and, then, the anti-virus code has to update its signature database very frequently to be able to catch the new virus. The main difficulty remain in the update procedures, because the new malware need to be analyzed precisely and the new signatures need to be created and distributed as soon as possible in order to control the propagation.

We propose a new approach to detect and identify malware by using formal methods involving static program analysis [2, 3, 4, 5, 6, 7, 8, 9], decision procedure [10, 11, 12, 13, 14, 15, 16, 17], and program verification techniques [18, 19, 20, 21, 22, 23, 24]. In this work, we adapt formal methods currently used to verify and prove systems correctness in order to provide new theoretical basis for the design of malware analyzer. We summarize our contributions as follows:

- *Formal program modeling*:
  We present several formal program models and show they are sound and efficient computationl model for the suspicious program under inspection. Those models are *abstract call graphs* and *pushdown systems* enriched with invariants at system calls cites.

- *Formal signature generation*:
  We provide new effective approaches in order to generates formal specifications that can be used as semantic aware signatures capturing the malicious behavior of the malware being analysed. Importantly, these malware formal specifications remains unchanged for most of obfuscated version of the code. In other words, most of the new versions of the malicious piece of code, obtained by obfuscation techniques, would still share the same formal signature.

- *Reduction of the malware detection problem to standard formal problems*:
  Considering each of our formal malware specification and each of our program models we were able to reduce the problem of malware detection to classical formal problems that can directly be addressed by existing formal methods and tools. For each reductions, we identify and express the obtained theoretical problems in standard formal

terms allowing the direct application of existing formal methods and associated tools. This point could be seen as one of the main contribution.

- *Formal intrusion detection*:
  Using such static analysis platforms and automatically generated invariants we show how to construct a suitable architecture for *Host-based intrusion detection systems*. Our model captures the control and data flow structure (i.e, accurate stream of system calls guarded by invariants). Any deviation reported by our monitor provide comes with the mathematicall proof of a violation of an application invariant.

Section 2 presents an overview of formal methods that are at the heart of our formal appraoches. Section 3 introduces malwares, their characterisation properties and obfuscation techniques. Section 4 shows that program invariants can be crucial for the construction of semantic aware malware analyzer. Section 4.2 provide important computational models for the analysis of programs. The important Section 5 provides our formal malware specification and the reduction of the malware detectin problem to problem that standart in the formal methods reseach community. Section 6 presents formal models for intrusion detections.

2 Formal Methods

We discuss key research areas providing an overview of formal methods that are at the heart of this dissertations and its contributions.

Formal methods aim at modeling and analysing systems using methods derived from, or defined by, underlying mathematically-precise concepts and their associated algorithmic foundations. By modeling we mean building specifications expressed in a particular logic, design or code. By analysing we mean the verification, or falsification, of system properties specified in an appropriate formal system.

Formal methods research aims at discovering mathematical techniques and developing their associated algorithms to establish the *correctness* of software, hardware, concurrent systems, embedded systems or hybrid systems, i.e. to prove that the considered systems are faithful to their specification. On large or infinite systems, or even systems with a huge or infinite numbers of reachable states, establishing *total* correctness is usually not practically possible. That is why we narrow our interest to safety and liveness properties that any well behaved engineered systems must guarantee. For instance, by using *static program analysis*, one could prove a software free of defects, such as buffer overflow or segmentation fault, which are safety properties, or non-termination, which is a liveness property.

In this sense, **static analysis** is used to generate *invariant properties*, which are assertions that hold true at a specific location on every possible run of the system. Thus static analysis can provide provable guarantees that even the most exhaustive and rigorous testing methods could not attain. Next, we discuss some proeminent techniques that proved useful in program verification.

2.1 Model Checking

Model checking [25, 26] is a verification technique for finite state systems. All states, together with all possible interactions in all possible runs of the system, are exhaustively enumerated. This could lead to huge structures. The main advantage of this method is that these structures can be build in memory *automatically*, from specifications in higher level languages. Several properties could then be algorithmically checked in this *state space*, by automatically checking correctness conditions at each state. In practice, we would *explicitly* explore all the state space to see if a specified unsafe property holds, that is, a "bad" reachable state exists. Such techniques have been used to prove correctness of hardware designs.

But this methods face the "state space explosion" problem, when it is not possible to hold all the state space even in a huge memory. As an alternative, one could use *symbolic* representations.

Symbolic model checking [27] comprise model checking techniques that use *symbolic* representations of sets of states. For instance, one could use logic formulae to represent a set of states. In this way, one could represent the set of reachable states in a very concise way. Moreover, binary decision diagrams (BDDs) [28] provide a very concise way to represent Boolean expressions and so can be used to economically represent sets of states in memory. We could also symbolically, represent infinite set of states by using, for example, semi-linear forms written as Presburger formulae. But even with these "nice" states space representations one could not completely deal with truly *infinite* systems.

Bounded model checking [29] is a model checking technique that exhaustively, or symbolically, analyses finite instances of infinite state systems. Any assertions that hold in a finite instance will hold on an infinite, i.e. more concrete instance. The basic idea in bounded model checking is to search for a counterexample in executions whose length is bounded by some integer k. If no defect is found then one increases k until either a defect is found, or the problem becomes intractable, or some pre-known upper bound is reached.

Model checking is often also used in falsification tests, i.e. for finding logical errors, rather than in verification, i.e. proving that errors do not exist. Here, in order to deal with infinite state systems, one needs to define a suitable *abstraction* of the systems regarding the properties one is looking for.

2.2 Abstraction

Abstraction [30] is commonly used in formal methods in order to achieve termination in the verification process for infinite systems, thus implying a trade off between termination of the verification process and completeness. Abstraction techniques first simplify the system in order to perform easier proof steps and later transfer the result back to the concrete system.

Abstract interpretation [30, 31] approaches depend on iteration and fixed point computations. Basically, it performs an approximate symbolic execution of a program or system until an assertion is reached that will remain unchanged along further executions of the program. However, in order to guarantee termination, the method introduces imprecision by the use of extrapolation operators called widening or narrowing. These operators often cause the technique to produce a too coarse abstraction that gives rise to weak invariants. Moreover, it requires manual intervention, as abstraction operations have to be provided and proved correct. Also, the right widening and narrowing operators have to be manually provided, which becomes a key challenge for abstract interpretation based techniques.

Predicate abstraction [2] requires a given set of abstraction predicates and an infinite state systems. It returns an "abstract state graph", in the form of a conservative finite state abstraction. Every execution in the concrete system has a corresponding execution in the abstract system. First the abstract version of the safety property is model-checked in the abstract system. If the property holds in the abstract system, then it holds in the concrete system too. Otherwise, an abstract counter-example trace is generated [32]. But it is not guaranteed that there is a concrete counter-example associated with the abstract one. It is possible that this abstract counter-example is a spurious one due to a too coarse abstraction, because the set of abstraction predicates induced a too coarse abstraction. On the other hand, if there is a concrete counter-example corresponding to the abstract trace, then we have generated the trace of a real defect in the original design. One can then automatically analyze the counter-example to find a real defect in the system, or one can refine the abstraction by adding new discovered predicates [33, 34, 35]. These steps can be done using *theorem provers* [10, 11, 12] or other decision procedures such as those stemming from *sat modulo theory* [36, 17].

2.3 Theorem Proving

Theorem proving [12, 11] is an interactive verification technique where the correctness condition of the system is written as a theorem in a fixed theory. A theorem is then proved by interleaving automatic and manual interventions, using inference rules of the theory, its axiomatization, and other proved deductions, i.e. previous lemmas. Theorem proving has all the methods of logic and mathematics at its disposal, which makes theorem proving tools very powerful. However, the complexity of the problems that can be addressed by current theorem provers is limited by the fact that they require manual intervention. Theorem provers have seen trendemous progress, and some recent techniques propose model checking frameworks using automated deductions, as in sequent analysis [37].

2.4 Invariant Generation

Safety properties can be proved by *induction* techniques in infinite state systems [38]. An *inductive invariant* must hold in the initial state of the system and every possible transition must preserve it. The former is known as the initiation condition and the latter is called the consecution condition. Actually, the verification problem of safety properties can be reduced to the problem of invariant generation. In other words, if it holds in a given state then it continues to hold in all of its successor states. Let φ_P be the desired property and denote by φ_{Inv} the inductive invariant obtained. If $\varphi_{Inv} \Rightarrow \varphi_P$ holds then the proof of φ_P is complete.

Automated inductive invariant generation is the essential step in proving safety properties. For example when proving that an application is free of bugs like division by zero, outbounds of arrays, buffer overflows, null pointer de-referentiation, and many others. Or in proving liveness properties such as progress and termination.

We know that the weakest precondition method [39, 40] and the Floyd-Hoare [40, 41] inductive assertion technique require loop invariants to establish total correctness. In order to be completely automatic these methods require the use of invariant generation techniques. Also, ranking function techniques [38] depend on automated invariant generation methods in order to prove termination. We could list here many verification approaches that are only practical depending on the easy with which invariant can be automatically generated. Verification diagrams [42] is another example.

We can separate invariant generation methods in two main classes. We have methods that are *goal-oriented* [43, 44, 45, 3] and follow a *top-down* approach. These approaches start with a candidate potential invariant which can be seen as a target property that implies the properties we want to prove. On the other hand, we find methods which generate invariants directly from the program code. These approaches [46, 30, 47, 48, 4] are *bottom-up* techniques.

3 Malware Characterisation

A *malware* is a program that has malicious intent. Examples of such programs include viruses, Trojans horses, and worms. These malicious intent display the following behavior, by:

1. following *infection strategies*,
2. executing a set of malicious actions, *payloads*,
3. evaluating some boolean control conditions, called *triggers*, to determine when a *payload* will be activated.

A classification of malware with respect to their effects and propagation methods is proposed in G. McGraw and G. Morrisett [49]. Also, L. M. Adelman [50] and F.B. Cohen [51] show that the research security community will deeply need a mathematical formalism that could serve as a scientific basis for their classification. We can distinguish three properties that characterise a class of malware:

- *Active propagation*: A malware can propagate passively or actively using self-instance replication or self-modification.
- *Population evolution*: A malware can be characterized by the evolution of the number of malware instances.
- *Context dependence*: in order to perform its malicious intent, a malware can depend on external context, *e.g.* it could require other executable code or a pre-compilation step.

Using these three properties, one could classify several types of malwares. For instance, *Virus* has an active propagation, several malware instances requiring external context. A *Worm* would have the same characterisation except that it is context-free. *Spyware* and *Adware* do not use replication or self-modification, they is no other malwares instances and do not depend on external execution or pre-compilation step. *Rabbit* has an active propagation, no population evolution and is contrxt-free. *LogicBomb* is partially context-free. Other hybrid types and network-based denial-of-service types like *botnets* and *zombie net-works*, ... could be also considered by combining and extending these properties. We could list more characterisation properties, like obfuscation and encryptions techniques, that we shall consider in our approaches for automated malware analysis.

3.1 Identifying Malware Concealment Behaviours

Current malware detectors are "*signature-based*" and equipped with a data base. These malware detectors are based on sound methods, that is, if the executable matches byte-signatures, then they guarantee the presence of the malicious behavior. They are equipped with a database of regular expressions that specify byte or instructions sequences that are considered malicious. But the main problem resides in the fact that these methods have a low degree of completeness.

Malware writers can then evade detection by current syntactic signatures and pattern matching approach by *injecting code into malwares in such a way that it preserves malicious behavior and makes the previous signature irrelevant*, because the regular expressions would not match the modified binaries. These techniques are called *obfuscation* [1]. Hackers frequently use obfuscation techniques to morph malware, as they are easy to write and very challenging to detect. Malware detectors, commercial anti-virus and scanners are susceptible to these techniques.

As common obfuscation techniques, one could cite *polymorphism* and *metamorphism.* In these methods, a virus encrypts, or obfuscates, its malicious payload and decrypts, or deobfuscates, it during execution. By doing so, a virus can succeed in morphing itself. Here the words "encryption" and "decryption" do not refer to cryptography, they are better thought of as obfuscation and deobfuscation strategies.

Once a new type of malware appears, the number of derivative malwares generated by obfuscation increases exponentially. Malware writers can easily generate new undetected viruses. As a consequence, the anti-virus code has to update its signature database very frequently in order to be able to catch the new virus the next time it appears.

3.2 Obfuscation Strategies for Infection Mechanisms, Triggers and Payloads

Malware can be obfuscated, i.e. its three body parts, namely Infection Mechanisms, Triggers and Payloads, can be first set in an encrypted/obfuscated form to avoid detection.

But it can not be entirely encrypted to be executable. It needs a *decryptor loop*, which deobfuscates its body parts in a specific writable memory location. These decryptor loop can use a random key and memory location that vary. Also, if it use cryptographic algorithms at this step then the encrypted part would be very difficult to detect. That is why anti-virus, and polymorphic malware detectors concentrate on detecting the decryptor loop.

To avoid detection, a *polymorphic* virus modify their decryptor loop at each instance using several automatic transformation, since a virus could easily generate billion of loop versions [52]. And *metamorphic* viruses change their body parts using a several obfuscation techniques when they replicate.

To modify the loop, or to obfuscate a body part, a malware uses a *mutation engine* [1] which can rewrite the loop with other semantically equivalent sequences of instructions [53] and rename register or memory locations. This transforms the original sequence of instructions into an equivalent one. In general this is done by using unconditional jumps, inlining and outlining the body of function codes, using new function calls, inserting junk code, using threaded versions, ... and a host of similar trirs. But these decryptor loops and the derived ones still share some common invariants properties, and, these invariants are more difficult to morph in an automated fashion.

4 Formal Models for Malware's Behaviors Identifications

4.1 Malware Invariant Generation

4.1.1 Malware Invariant as Semantic Aware Signature

In [54, 55, 56], malware-detection algorithms incorporate instruction semantics in order to detect malicious behavior. Semantic properties are more difficult to morph in an automated fashion. But the main problem of these approaches is that they relay on too coarse abstracted semantic information e.g. *def-use* information. Instead of dealing with regular expressions, they match a control flow graph, enriched with def-use informations to the vulnerable binary code. Those methods eliminate some techniques of obfuscation but it is still simple to obfuscate def-use information by adding any junk code or by reordering operations that would either redefine or use the variables present in the def-use properties.

Our approach to the problem of malware detection propose to generates invariants directly from the specified malware code, and use it as *semantic-aware* signatures that we call *malware-invariants*. Now consider a suspicious code. We could check if there is one assertion in the malware-invariant data base that holds in one of the reachable program states. In order to do so, we can use formal methods for *invariant generation* and *assertion checking*. This will complicate, make it much more for a hacker to evade the detection using common obfuscation techniques. Thus, for each family of viruses we would have few semantic aware signatures. We propose to use, combine and compose many static and dynamic tools in order to automatically generates invariants which are semantic-aware signatures of malwares, i.e. they are malware-invariant.

4.1.2 Automatic Generation of Malware Invariants

We say that an analysis is *static* when it does not run the application. Anti-virus use a data base of signatures and a *scanning* algorithms to look efficiently for several patterns at a time. Each of these patterns represents several different signatures. As we saw in the previous sections, present malware writers can evade such pattern-matching techniques.

Malware invariants could be computed directly using our invariant generations methods, together with any other invariants computed using different techniques, such as based on a complementary logic. Several tools could be made available though a *communications framework*. We provide a theoretical basis for the construction of static analysis platforms that can form a suitable architecture for the extraction and identification of possible malicious behaviours. We propose to consider several invariant generations techniques where the automatically computed invariants would be express in specific logic. Considering the methods described in [5], one can handle logic with non-linear arithmetic and inequalities. Using them one can generate non-linear invariants in several guises like assertions, formulae over inter-relationship values of registers, memory locations, variables, system call attributes, and similar others.

Example 1. *Consider the following piece of code expressed in an intermediate representation after a transformation process.*

```
integer R_1, eax, ebx;
...
where (eax=R_1 && ebx=1)
 ...
while (eax>ebx){
    eax = eax * ebx + eax;
    ebx = ebx * ebx;
  }
```

We can, for instance generate the following invariant

$$R_1(1 - R_1)eax^2 + eax * ebx - eax + ebx^2 - 2 * ebx + 1 = 0.$$

□

In order to handle systems and function calls effectively, we can consider the techniques described in A. Tiwari et al. [18, 6] which provide invariants over a logic with uninterpreted function and inequality. This theory uses several level of Abstract Interpretation [31] and is based on Unification Theory [57]. Moreover the implementation also requires a modifications to adapt it of the CIL tool [58].

Example 2. *Consider the following piece of code with uninterpreted* system calls*. Note that it could be any other systems calls with a similar signature.*

```
  void
  F_D(int fd_1,int fd_2){
  ...
  int t1,t2,t3,t4,t5,a;

  if (fd_1==fd_2) {
     t1=dup(fd_1-fd_2);
     t2=dup(dup(0));
     }
  else {
        t2=dup(t1);
        }
    t3=dup(t1);
    a=fd_2-1;
    if (fd_1==a){
        t4=dup2(fd_1,2*fd_1+1);
        t5=dup2(fd_1,dup2(fd_1,2*fd_2-1));
        }
    else {
        t5 = dup2(fd_1, t4);
         }
    }
```

Using [18, 6] we can generate the following invariant

```
Invariant Generation [Logic: Uninterpreted Function]
t3 = t2
fd_2 + -1.000000*a = 1.000000
t2 = dup(t1)
t5 = dup2(fd_1, t4)
```

As we can see it yields inter-relationships between system calls returns values and attributes. □

The static analysis provided in the Apron library [7, 59] infers inequality invariants of the form $a \leq b + c \leq d$ at each program location.

Example 3. *The following example is borrowed from [60] in order to illustrate the Apron numerical abstract domain library [59].*

```
Annotated program after forward analysis
proc incr (x : int) returns (y : int) var ;
begin
  /* (L3 C5) [|-x+10>=0; x>=0|] */
  y = x + 1; /* (L4 C10)
                [|-x+y-1=0; -x+10>=0; x>=0|] */
end

var i : int;
begin
  /* (L8 C5) top */
  i = 0; /* (L9 C8) [|-i+11>=0; i>=0|] */
  while i <= 10 do
    /* (L10 C18) [|-i+10>=0; i>=0|] */
    i = incr(i); /* (L11 C16)
                    [|-i+11>=0; i-1>=0|] */
  done; /* (L12 C7) [|i-11=0|] */
end
```

At each program location, Apron [59] added in comments inequality invariants. In this example the numerical abstract domain was a convex polyhedra. □

We could as well, consider any other logic which has an associated invariant generation techniques and tools. The main contribution here is that several obfuscation techniques presented in section 3.2 will not change the computed invariants.

4.1.3 Quasi-Static Analysis for Malware Invariants

In contrast, we propose new concepts and a theoretical framework that can be used to compute exact invariants, i.e. provable invariant, using hypotheses, i.e. likely invariants, that are true properties on all observed execution traces, in a certain training period. We could then generate likely invariants which are properties that hold at any program point in the observed execution trace, using test methods. We call this new view of program analysis "*Quasi*-Static Analysis". Figure 1 illustrates the architectural structure of this new theoretical framework. Then we could turn likely invariants into invariants using verification methods, like assertion checkers [61, 20].

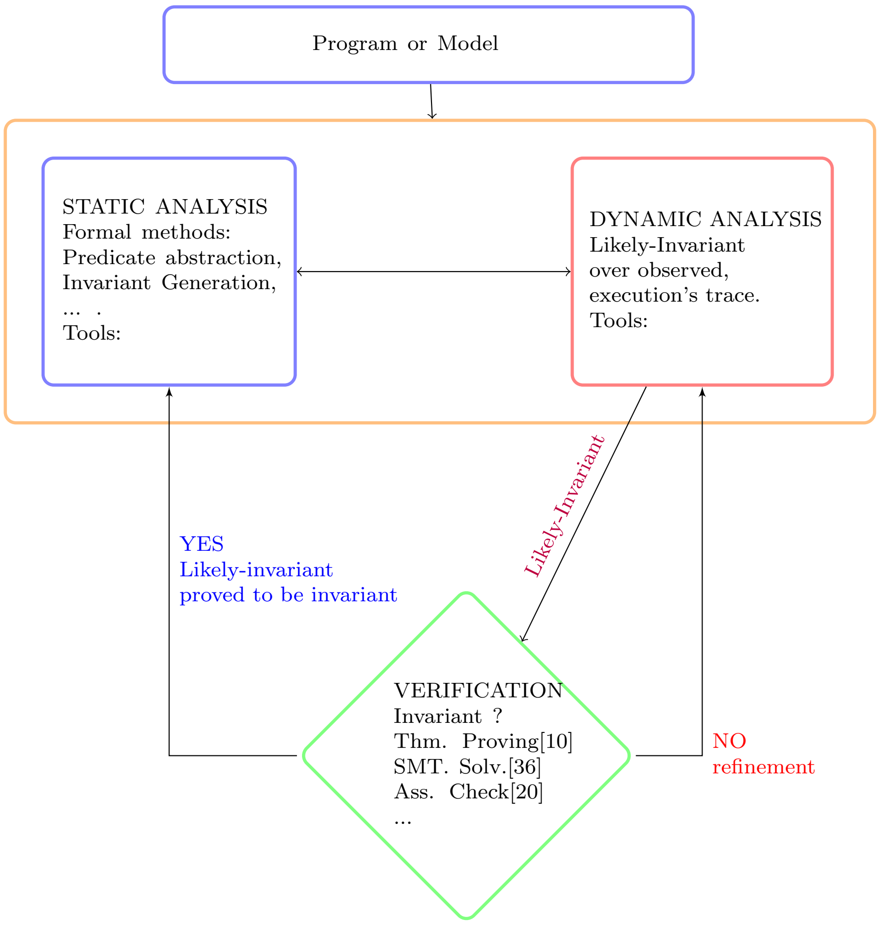


Figure 1: Quasi-static invariant analysis

Some of the likely-invariants computed by a Dynamic Analysis are real invariants. They hold in all possible executions of the program. Then, using theorem provers or assertions checkers, for instance one could check if the proposed properties during the Dynamic analysis are real invariants. This framework would incorporate any automated invariant generation techniques (e.g., [3, 4, 5, 6, 7, 62, 63, 8, 9].). The formal metods community have provided several techniques for assertions checking (e.g., [18, 19, 20, 21, 22]) and theorem provers (e.g., [10, 11, 12]) and decision procedures (e.g., [13, 14, 15, 16, 17]). Also, there exists effective tools using a dynamic program analysis, like Daikon [64], generating large numbers of interesting likely-invariant. If we consider binary code, we can use intermediate representation tools. In [23, 24], the authors introduce Vine IL an intermediate language for reasoning about x86 assembly. It translates accurately all x86 instructions in a simple, RISC like language. Moreover, they provide the translation of there intermediate language to C. Also, we note that Vine [23, 24] converts its intermediate language in a Dijkstra's Guarded Command Language [65] allowing the computation of its weakest preconditions (in CVCLite [15] format) that we can use as invariants too.

In the following section we show how these malware invariants are then treated as malware signatures.

4.2 State-Based Models for Malware Analysis

In this section we use automata as computational model for malware programs. To be able to reason directly from unknown vulnerable binary code, one needs an intermediate representation [66, 67] for the binary code. In our first models we consider that our analysis depart from a pre-established intermediate representation rewritting the semantics in terms of imperative instructions.

First we propose an intra-procedural analysis to define model a procedure. This model is the abstract state graph of the procedure. It is a much more precise version of the control flow graph of the procedure.

Definition 4.1. ***ASG: Abstract State Graph.*** *Let P be a procedure, and let $p_1, \ldots, p_k$ be a set of arbitrary predicates over P's program variables. An ASG of P using the predicates $p_1, \ldots, p_k$ is the intraprocedural control flow graph*

$$G = \langle V, V_a, Enter_p, Exit_P, \delta_a \rangle$$

associated with the program source code of P where:

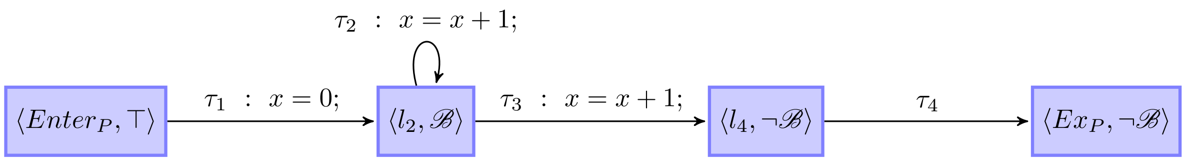
- *V is the finite set of program locations,*
- *$Enter_P \in V$ is the entry point of P*
- *$Exit_P \subseteq V$ is the finite set of P's exit points.*
- *V_a is the finite set of* abstract states. *Each such abstract state is a pair (v, e) where v is a program control location and e a valuation of the predicates $p_1, \ldots, p_k$. For instance for $k = 3$, a node $pc1, (0, 1, 1)$ indicates that at location $pc1$, the predicate $P_1 \wedge \neg P_2 \wedge \neg P_3$ is true.*

- *$\delta_a \subseteq V \times V$ is a transition relation.* □

Example 4. *Consider the following program:*

```
x=0;
While (x <= n){
   x=x+1;
}
```

Consider the predicate $\mathscr{B} \equiv (x \leq n)$, its abstract state graph is given below:


$\tau_2 \; : \; x = x + 1;$
$\langle Enter_P, \top \rangle$
$\tau_1 \; : \; x = 0;$
$\langle l_2, \mathscr{B} \rangle$
$\tau_3 \; : \; x = x + 1;$
$\langle l_4, \neg\mathscr{B} \rangle$
τ_4
$\langle Ex_P, \neg\mathscr{B} \rangle$

with $V = \{Enter_P, l_2, l_4, Ex_P\}$, $Exit_P = \{Ex_P\}$, and $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$. □

The ASG of a procedure is turned into a Guarded Control Flow Graph by guarding transitions originating from an abstract state (v, e) by the Boolean combination of the predicates $p_1, \dots, p_k$ given by the valuation e. This Boolean combination is an invariant of the program at location v.

Definition 4.2. ***(PCFG: Predicate Control Flow Graph)*** *A PCFG of a procedure P is the* intraprocedural *control flow graph $G_P = \langle V, v_0, Exit_P, \mathcal{G}, \delta_{lP} \rangle$ associated with the program source code of P where:*

- *V is the finite set of program locations.*
- *$Enter_P \in V$ is the entry point of P.*
- *$Exit_P \subseteq V$ is the finite set of P's exit points.*
- *$\delta_P(V \times \mathcal{G}) \subseteq 2^{\mathcal{I}} \times V$ is a transition function where a block of instructions in $\mathcal{I}$ is guarded by an invariant in the set $\mathcal{G}$.* □

Our model abstracts away from the PCFG any transitions and instruction blocks that do not refer to a system call or to function call. The result is a predicate call graph.

Model 1. ***(PCG: Predicate Call Graph)*** *Let P be a procedure and let its PCFG be*

$$G_P = \langle V, Enter_P, Exit_P, \mathcal{G}, \delta_{lP} \rangle.$$

A predicate call graph (PCG) of P is a tuple

$$G = \langle \mathcal{S}, Enter_P, Exit_P, \delta_P, \lambda, FuncC, SysC \rangle$$

where :

- *$\mathcal{S}$ is a finite set of states.*

- *$FuncC$ is the finite set of* function call *that appear in P.*
- *$SysC$ is the finite set of* system call *that appear in P.*
- *$\lambda : \mathcal{S} \rightarrow 2^V$ labels the state with a set of consecutive program points that identify a basic block, i.e. straight-line pieces of code without any jumps nor system or function calls.*
- *Finally,*
$$\delta_P \subseteq (\mathcal{S} \times \mathcal{G} \times (SysC \cup FuncC) \times \mathcal{S})$$
is a transition relation. □

Intuitively, these action guarded transitions are labeled by a system ($SysC$) or a function ($FuncC$) call that occur at the last program point labeled in the current state (*basic block*). They should satisfy:

- If there is a function call $F \in FuncC$ at a program point $v \in V$ then $\exists s, \exists s' \in \mathcal{S}$ and $\exists g \in \mathcal{G}$ s.t. $v \in \lambda(s)$, and $\delta_P(s, g) = (f, s')$.
- If there is a system call $C \in SysC$ at a program point $v \in V$ then $\exists s, \exists s' \in \mathcal{S}$ and $\exists g \in \mathcal{G}$ s.t. $v \in \lambda(s)$, and $\delta_P(s, g) = (c, a, s')$.

Such model allows the representation of executions involving functions/systems calls where the return address has been modified (i.e., not matching the calls site location, ...). Such type of non-standard running behavior is often followed by malicious codes.

In the next model, we do not consider intermediate representation and we provide a computational model for (malicious) binary code. Our approach still depart from an abstract state graph considering predicates allowing the sound summarization of instructions block that do not modificates the progam's stack of calls sequences. We use a *push down system* in order to model the "`puch`" and "`pop`" instructions.

Model 2. ***(PPDS: Predicate Push Down System)*** *Let P be a procedure and let its Predicate Call Graph PCG be*

$$G_P = \langle \mathcal{S}, Enter_P, Exit_P, \delta_P, \lambda, FuncC, SysC \rangle.$$

Predicate Push Down System(PPDS) of P extends G_P with a finite stack alphabet

$$\Gamma \subseteq \mathbb{D},$$

where $\mathbb{D}$ is a finite domain and $\gamma_0 \in \Gamma^*$ the initial stack configuration. $\lambda : \mathcal{S} \rightarrow 2^V$ labels the state with a set of consecutive program points that identify a basic block without "`pop`" or "`push`" assembly instructions. We define the transition system δ with the set of actions A as follow:

- *A_E is the finite set of actions given by the following grammar:*
$$a \leftarrow \mathsf{Push}(\gamma) | \mathsf{Pop}(\gamma) | \epsilon,$$
where $\gamma \in \Gamma$ and $\mathsf{Push}(\gamma), \mathsf{Pop}(\gamma)$ are the two common operations that pop and push γ into the stack.

- *Finally,*

$$\delta \subseteq \cup(\mathcal{S} \times \Gamma \times \mathcal{G} \times (SysC \cup FuncC \cup \{\epsilon\}) \times (A \cup \{\epsilon\}) \times \mathcal{S} \times \Gamma).$$

□

A *configuration* of P is $(s, \gamma) \in \mathcal{S} \times \Gamma^*$. A possible run is modeled by a sequence of configurstions

$$\rho = (s_0 \rho_0), \ldots, (s_k, \rho_k),$$

where for every $s_i \in \mathcal{S}, \rho_i \in \Gamma^*$, $\rho_0 = \epsilon$ and for every $1 \leq i \leq k$ the following rules apply:

- *Making a* `call`*:*
  If $(s_i, \epsilon, g, c_i, \epsilon, s_{i+1}, \epsilon) \in \delta$ then the call a_i is allowed by our model when transitioning from s_i to s_{i+1} if the guard $g \in \mathcal{G}$ evaluates to *True* at s_i and there is no stack activity $(\rho_{i+1} = \rho_i)$.

- *Assembly instruction* `push`*:*
  If $(s_i, \epsilon, g, \epsilon, \mathsf{Push}(\gamma), s_{i+1}, \gamma) \in \delta_E$ then if the guard g evaluates to *True* at s_i, γ is *pushed* onto the stacks, meaning that $\rho_{i+1} = \gamma \cdot \rho_i$, with s_i interpreting the call site mentioned in γ, and where "$\cdot$" is the operator of concatenation.

- *Assembly instruction* `pop`*:*
  If $(s_i, \gamma, g, \epsilon, \mathsf{Pop}(\gamma), s_{i+1}, \epsilon) \in \delta$, then, it means that the program pops γ from the program's stack under the condition that g is true. Meaning that if $\rho_i = \gamma\omega$ then $\rho_{i+1} = \omega$ with $\omega \in \Gamma^*$. □

Model 2 could be seen as an extension of the push down system proposed in [68] as in our model, the program's stack manipulations, the function and system calls are guarded by invariants.

Theorem 4.1. (**Expressivity**)*:*
Let P be a procedure and let G its PPDS (See Model 2). Then G can be simulated by the pushdown system provide in [68].

Proof. Let P be a procedure and let G be its PPDS and G' be the push down system from [68], where the finite transition can be written as $\Delta \subseteq (V \times \Gamma) \times (V \times \Gamma^*)$ in our notations. Now, we show that G can be simulated by G' with the following encoding. Consider two consecutive configurations $(s_i, \rho_i), (s_{i+1}, \rho_{i+1})$, in a possible path of G, obtained by taking the transition $\tau \in \delta$.

- If $\tau = (s_i, \epsilon, g, \epsilon, \mathsf{Push}(\gamma), s_{i+1}, \gamma) \in \delta$, then in G', there exists a transition $((v_j, \gamma), (v_k, \rho_{i+1})$ where $(v_j, v_k) \in (\lambda(s_i), \lambda(s_{i+1}))$.

- If $\tau = (s_i, \gamma, g, \epsilon, \mathsf{Pop}(\gamma), s_{i+1}, \epsilon) \in \delta$, then in G' there exists a transition departing from $v_j \in \lambda(s_i)$ poping γ from the top the stack in G' and reaching a location $v_k \in \lambda(s_{i+1})$.

- If $\tau = (s_i, \epsilon, g, c_i, \epsilon, s_{i+1}, \epsilon) \in \delta$ then $\rho_{i+1} = \rho_i$ as there is no stack activity. And in G' there exists a transition $((v_i, \rho_i), (v_k, \rho_i))$ with $(v_j, v_k) \in (\lambda(s_i), \lambda(s_{i+1}))$.

□

Corollary 4.1. (**Precision**)*:*
Our two models are primary based on an abstract state graph, i.e., a boolean program, that could be build using any type of predicate abstraction techniques (see Section 2.2 paragraph ***Predicate abstraction****) adding any discovered predicated from several effective decision procedures [33, 34, 35] and SMT solvers [36, 17]. This extension would provide more precise identifications and specifications of malware behaviors.*

5 Formal Methods for Malware Classification and Detection

In the following Section 5.1 we provide several formal ways to specify the malicious behavior that can be treated as the core of the semantic of the considered malware. In section 5.2, with both the specification methods and the models introduced we are able to reduce the problem of malware classification and detection to classical problems very standard for the formal method research community.

5.1 Formal Specification and Malware Behavior

In the previous section we provided static analysis and specific computational models with there properties shown to be central for the modelization of suspicious codes. Here we introduce several formal computational ways to specify malicious behavior. By using formal methods, we propose to extract and summarize the semantic of the malicious behavior of the malware. Now, considering a suspicious code, these specification could be used to check if the considered program allow the execution of malicious behavior. This point is illustrated in Figure 2.

Here, we provide different types of formal signatures for malware behaviors.

- **Malware invariant**: As we saw in Section 4.1, invariant expressed in a logic related to the semantic of the malicious code can be considered as very effective malware signature. Also, in Section 4.1.3, we provide the theoretical foundation of a framework allowing the combination of static and dynamic analysis in order to generate more invariants. Following such desing, one could obtain a formal method tool-bus where several approach dealing with different logic could end up generating a larger set of invariants.

- **Malware Abstract Call Graph**: Our Model 1 can be seen as an abstract state graph augemented with invariants at function and system calls cites. In Section 4.2, we saw that those automaton are very efficient computational model for programs. More other we saw that there are several formal methods approaches and tools that allow there automatic generation, refinements and analysis (predicate abstraction, decision procedures, assertion checker, ...). Here, we will consider these PCG automata as

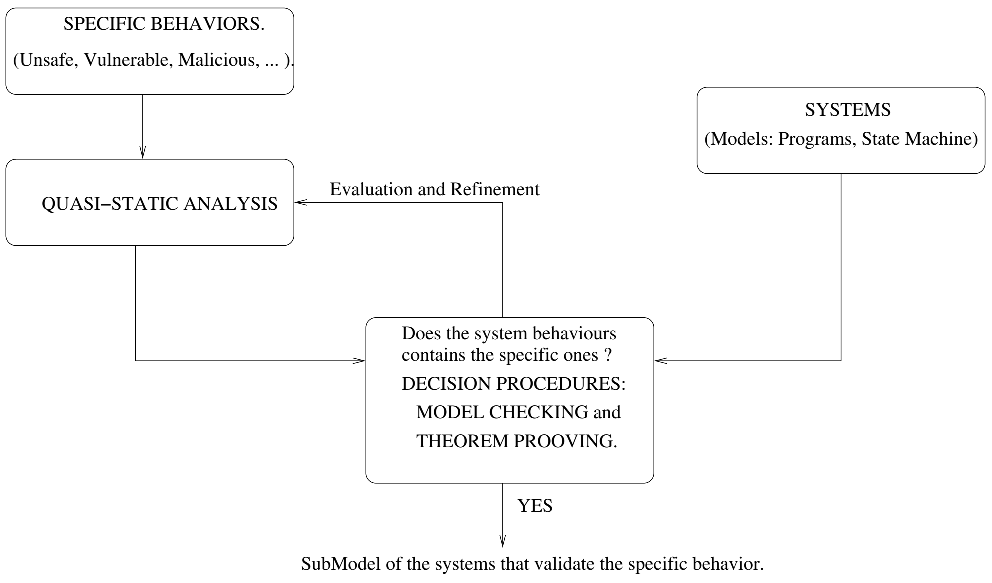

SPECIFIC BEHAVIORS.
(Unsafe, Vulnerable, Malicious, ...).
SYSTEMS
(Models: Programs, State Machine)
QUASI–STATIC ANALYSIS
Evaluation and Refinement
Does the system behaviours contains the specific ones ?
DECISION PROCEDURES:
MODEL CHECKING and
THEOREM PROOVING.
YES
SubModel of the systems that validate the specific behavior.

Figure 2: New formal methods to verify if the considered systems contains common behavior with another specified systems

signature. These signature would embeed control and data flow informations, which would allow use to describe malware behaviour in a more precise and automatic way.

- **Malware Pushdown Systems**: In Section 4.2 we also provided the PPDS Model 2 in order to first be able to modelize the program's stack. Moreover we saw that is is a very effcient computational model for binary codes. Here, will also consider these pushdown system in order to extract the semantic of the central stack manipulations made once the malware operates.

- **Malware CTL formulas**: in order to be able to express specific behaviors on the malware's stack we use formula expressed in a branching-time temporal logic CTL [69, 70]. More precisely, we consider SCTPL specifications [68] as it extend the CTL logic with variables, quantifiers and predicates. This logic allows the expression of formula and predicates over the stack. In the following we borrow an example of SCTPL specification of email worm from [68].

Example 5. *Here we recall an email worm and its SCTPL specification provided in [68] for illustration of SCTPL formulas.*

– *Email worm:*

```
lea eax, [ebp + ExistingFileName]
push eax
push 0
call GetModuleFileNameA
```

```
...
lea eax, [ebp + ExistingFileName]
push eax
call CopyFileA
```

– *SCTPL specification:*
  $\phi_{email-worm} = \exists m(\exists r_0(\mathsf{EF}(lea(r_0,m) \wedge \mathsf{EX}\ \mathsf{E}[\neg\exists v(mov(r_0,v) \vee lea(r_0,v))$
  $\mathsf{U}(push(r_0) \wedge \mathsf{EX}\ \mathsf{E}[\neg(push(r_0) \wedge \exists v(pop(v) \wedge r_0\Gamma^*))\mathsf{U}(call(GetModuleFileNameA)$
  $\wedge 0\ r_0\Gamma^* \wedge \exists r_1(\mathsf{EF}(lea(r_1,m) \wedge \mathsf{EX}\ \mathsf{E}[\neg\exists v(mov(r_1,v) \vee lea(r_1,v))\mathsf{U}(push(r_1)$
  $\wedge \mathsf{EX}\ \mathsf{E}[\neg(push(r_1) \vee \exists v(pop(v) \wedge r_1\Gamma^*))\mathsf{U}call(CopyFileA) \wedge r_1\Gamma^*])]))))])]))))$.

The formula $\phi_{email-worm}$ represent precisely the semantic of the malware behavior depicted in the binary code above. The reading of the formula requires some familiarity with CTL temporal operators (E,X, F, U) and there semantics. We note the presence of predicates over the stack, (e.g., $r_0\Gamma^*$ meaning that the top of the stack contains the value of register r_0.). The formula models first that GetModuleFileNameA is called with the parameters 0 and the memory location m (i.e., [ebp + ExistingFileName]). In other words, at the call cite of GetModuleFileNameA, the stack is of the form $0m\Gamma^*$. Thus, the worm write its executable name at address m. Then the worm copy its files in other location by using CopyFileA with m as parameters. In other words after executing the instructions (GetModuleFileNameA(0,m)) the worm call at some point later CopyFileA(m) and to do so, m has to be at the top of the stack. All details could be found at [68]. □

Those SCTPL formula are specified manually but they are very useful to express precisely an identified malicious behavior. In [68], they specifed several identifier behavior of real world malwares. We note that the other specification techniques presented above are equipped with automated formal appraoches allowing there generations. □

These formal signatures carry more data and control flow information summarizing in a consice and sound way the malicious behaviour. As we noted in Section 4 such signature will be much harder to morph with the described obfuscations techniques. We recall that the number of derivative malwares generated by obfuscation increases exponentially. These formal signatures induce a new approach for malware classifications. We would much small database of formal signature. These formal signature are logic or state-based models. For each familly of malware could could generate the weakest preconditions of all associated malware invariants using theorem proving or deductions procedures. Also, considering all malware astract call graphs and PDS signatures associated to a specific family of malware, we could look for the smallest language obtain intersecting of all the languages generated by these automata-based signatures.

5.2 Reduction to Classical Formal Problems

By considering each of the presented models and specifications, we reduce the problem of malware detection in Section 5.2.1 and we provide theoretical framework to classify these detected malware. For instance, we could extract invariant from the code being inspected with similar mentioned methods and check, using decision procedures, if it implies one of the malware invariants. Or, we can check if the code behavior allow a possible run in one of the malware PDS. The malware detector method described in this Section is sound with respect to the signature that is being considered as the representation of malicious behavior and the computational models of the code being inspected.

5.2.1 Formal Methods for Detection

We combine the different formal specifications and models and reduce the detection problem to standard problem in the formal methods research community.

- **Reduction to theorem proving and decision procedure**:
  Consider a malware, we propose first to collect as much as possible invariants asa described in Section 4.1 in order to obtain a malware invariant ϕ_{sign}. Then one can use similar invariant generation methods, applied over a considered vulnerable executable code being inspected, and generate a global invariant ψ_{Exe}. Then, by using theorem prover or decision procedure like the one based on SMT solvers, one could check if

  $$\boxed{\psi_{Exe} \Rightarrow \phi_{sign}} \tag{1}$$

  – *Application*:
    In order to adress the problem 1 we could consider powerful automated tool from theorem proving and Satisfiability Modulo Theories (SMT) based decision procedures. We could use the theorem provers PVS [10], Isabelle [11] and/or Coq [12]. As for SMT-based decision procedures, we could consider STP [16] (over the theory of bit-vectors and arrays). CVC3 [15] is an automatic decision procedure for Satisfiability Modulo Theories (SMT) problems. We can used it to prove the validity or satisfiability) of first-order formulas in a large number of combined built-in logical theories. Also, Yices 2 [17, 71, 36] is complete and performant SMT solver that decides the satisfiability of formulas expressed with uninterpreted function symbols (that could represent function or system calls) with equality, linear arithmetic over the real and integer, tuples, bitvectors and scalar types. The fromal methods community have made tremendous efforts and reach great improvement in decision procedures and the SMT-LIB platform [13, 14] unify and integrate several approaches to analyse validity and satifiability of such types of constraints. Also, we note that Vine [23, 24] offers direct interface with STP [16] and its generated weakest preconditions are express in CVCLite [15] format. Thank to the problem reduction 1, we propose to check $\psi_{Exe} \Rightarrow \phi_{sign}$ by using mentioned Theorem provers and SMT-based decision procedures.

- **Reduction to Reachability analysis and Program Verification**:
  Here, we consider M a Abstract Call Graph (see Model 1) as computational model for the code under inspection. As specification for malicious behavior, we will use malware invariant ϕ_{sign} from pre-computed malware invariants data bases obtained following

the arquitecture proposed in Section 5.1. If the malware-invariants describe reachable states in the verification process, then we can guarantee that the software displays the suspicious behavior. We recall from definition 4.1 that M is build over *abstract state* (v, e) where v refers to a program location (a call cite s such that $v \in \lambda s$ according to the construction of Model 1). And e is a valuation of the predicates $p_1, \ldots, p_k$ used in construction M. The problem of detecting the malicious behavior specified by ϕ_{sign} in the operational semantic of considered program is reduce to the reachability problem of finding a reachable abstract state $a = (v, e)$ in M such that the valuation e implies that the formula ϕ_{sign} is evaluated true. In the notations introduced in definition 4.1, the problem of malware detection is in this case reduced to the following reachability problem where we have to check if :

$$\boxed{\exists (v, e) \in V_a \ : (e \models \phi_{sign})} \tag{2}$$

– *Application*:
  One can, then, use program verification tools and methods, to detect the presence of malicious behavior described by our malware-invariants. The formal methods community have provided a large number of efficient program verification tools (e.g., SATURN [20], BLAST [19], ASTRÉE [21], F-SOFT [22], CodeSonar from Grammatech Inc, PolySpace from PolySpace Technologies, Prevent from Coverity, ...) based on abstraction methods (Predicate abstraction [2], Abstract Interpretations [31], Assertion Checking [18]) that are capable of handling large programs and safety property being check and expressed in complex combined logics (aliasing, nonlinear arithmetics, ...). Here we could use the same tools and approaches but with malware abstract state graphs and malware invariants instead. Also, we note that most of the mentioned tools embeed there own abstract state graph model which facilitate the verification process of malware invariant against abstract call graph. For instance in [23, 24],

- **Reduction to CTL Model Checking for Pushdown Systems**:
Consider a malware CTL formulas ϕ_{CTL} (see Section 5.1) expressed in a SCTPL logic as formal signature for a specific malware behavior. Now, we consider pushdown systems (PDS) that we called PPDS and provided in Model (2) (see Section 4.2). Let M be the PPDS (see Model 2) of the suspicious code under inspection. If M allow an accepting path (i.e., a possible run) satisfying ϕ_{CTL} we have the proof that the code under inspection allows the malicious behavior expressed by the specification associated to ϕ_{CTL}. The malware detection problem can thus be interpreted as a CTL Model Checking problem for PPDS. Now, we follow standard automata theoretic approach for CTL Model-Checking. The formula ϕ_{CTL} is express in an extension of the CTL logic called SCTPL and introduced first in [68]. In [68], the authors show that one can compute a Symbolic Alternating Buchi Automata $A_{\phi_{CTL}}$ such that the language $\mathcal{L}(A_{\phi_{CTL}})$ of $A_{\phi_{CTL}}$ describe exactly all executions branching trees/runs satisfying ϕ_{CTL}. Our PPDS Model (2) is an extension of the pushdown systems proposed in [68] to abstract call graph augemented with invariants. As we saw in Section

4.2, the semantic of the stack operations are not compromised by this extension. Moreover our Theorem 4.1 shows that our PPDS Model 2 can be simulated by the PDS proposed in [68]. This guarantee the fact that our Model 2 can be translated into a Symbolic Alternating Buchi Automata using an adaptation of the algorithm used in the translation proposed in [68]. Thus, one can translate M into a Symbolic Alternating Buchi Automata A_M and consider its associated language $\mathcal{L}(A_M)$. In this case the malware detection problem can be seen as the following automata theoretical problem where we have to check if:

$$\boxed{\mathcal{L}(A_{\phi}) \cap \mathcal{L}(A_M) = \emptyset} \tag{3}$$

We can first perform the synchronized product $A_{\phi_{CTL}} \times A_M$ and then proceed to the emptiness check of the language $\mathcal{L}(A_{\phi_{CTL}} \times A_M)$. Thus the problem 3 can be rewritten as follow:

$$\boxed{\mathcal{L}(A_{\phi_{CTL}} \times A_M) = \emptyset} \tag{4}$$

We note that in [68], the authors presented a reduction of the malware detection problem to SCTPL model-checking for there introduced PDS. Here, we show that such reduction still holds when using our extended Model 2 that is augemented with abstract control and data flow information (e.g., function and system calls guarded by predicates, invariants).

- *Application*:
  We suggest to use and adapt the translation algorithm to Symbolic Alternating Buchi Automata, the product and the emptiness check algorithms provided in [68].

• **Reduction to synchronized product and language emptiness check**:
We suggest here to use our abstract predicate call graph provide by Model (1) in order to specify the malicious behavior. let A_{sign} be our malware abstract call graph that is used here as the signature of the malware. Now, we will also use these abstract call graph as computational model for the suspicious code under inspection. In other word, both the malware formal signature and the model use for the program under inspection are express wit the same computation Model (1). We denote by A_M the automata modelizing the program being inspected. Here, if there exists an acceptable path (i.e., a run reaching acceptable states) in A_M that is also an acceptable path in A_{sign} then we have the program under analysis allows the execution of a behavior modelized by the formal specification associated to A_{sign}. In other words, we reduce the malware detection problem to the following problem consisting in the verification of the condition:

$$\boxed{\mathcal{L}(A_{sign} \times A_M) = \emptyset} \tag{5}$$

Here, one need first to preform the synchronized product $A_{sign} \times A_M$ and the emptiness check of the language $\mathcal{L}(A_{sign} \times A_M)$. We note that we can use the PPDS Model (2) as formal signature and model for the analized code with the same automata-theoretical approach too.

- *Application*:
  We could use standard automata algorithms for the synchronized product and the emptiness check operations over Model (1). We could also use there adapted versions for PDS when considering Model (2) for the formal signature and the analyzed code. In [54, 55, 56] instead of dealing with malware signature based on regular expressions, they match a control flow graph, enriched with def-use informations to the vulnerable binary code. Here, we could adapt these matching techniques (register renaming, etc ...) to our abstract call graph carrying more data and control flow integrity.

□

In the following Section 6, we build a theoretical platform, and its associated Host-based intrusion detection systems, following similar a proof of concept.

6 Formal Metods For H-IDS

Standard Host-based intrusion detection systems (H-IDS) approaches [72, 73, 74, 75] monitor an application execution and report a violation at any deviation from its model. These H-IDS Looks for anomalies in the stream of system calls issued during the monitoring of the execution of the applications being protected. [76]. The authors of [72] suggested the importance of the use of static analysis in the construction of H-IDS monitors. We build our model using static analysis, i,e, using predicate abstraction, invariant generation and propagation techniques. Our model detects *mimicry attacks* [77, 78] and *non-control-data flow attacks* [79]. We build our model by annotating system call locations with a set of predicates and logic assertions that are invariants properties at the corresponding location of the application being protected. Such invariants yields inter-relationships between system calls arguments, system calls returns values, input/output values, branch predicates values, ... In order to assure *data flow integrity* our model checks the pre-computed invariants against the image of the process before any system call. Our model captures *control flow integrity*: system calls are allowed only if they respect the specific order pre-established. A deviation will be raised if one of the pre-computed invariants is not true at those system call locations or if the stream of system calls do not only occur in the order specified by the model. We use invariant generation and assertion checking techniques to build our model which can be described as a control flow graph in which system call locations are annotated with a set of predicates and logic assertions that have to remain true at those system call locations. Our model do not yield false alarm because when any reported deviation is based on a proved violation of a stacticaly computed invariant or on a proof of an abnormal order of system call execution in the monitored application. When determinism is reach in our abstract system call graph, we can also verifiy if the application is free of application logic bugs and vulnerabilities. In preliminary works [80], we introduced preliminary discussions on the properties of such model. Departing from the PCG (see Definition (4.2) in Section 4.2), our model abstracts away any transitions and instruction blocks that do not refer to a system call or to function call. Thus we obtain a guarded call graph.

Definition 6.1. ***(GCG: Guarded Call Graph)*** *Let P be a procedure and let its PCFG be*

$$G_P = \langle V, Enter_P, Exit_P, \mathcal{G}, \delta_{lP} \rangle.$$

A guarded call graph (GCG) of P is a tuple

$$G = \langle \mathcal{S}, Enter_P, Exit_P, \delta_P, \lambda, FuncC, SysC, Pred_{site} \rangle$$

where :

- *$\mathcal{S}$ is a finite set of states.*
- *$FuncC$ is the finite set of* function call *that appear in P.*
- *$SysC$ is the finite set of* system call *that appear in P.*
- *$Pred_{site}$ is a finite set of* predicates associated to function call sites.
- *$\lambda : \mathcal{S} \to 2^V$ labels the state with a set of consecutive program points that identify a basic block, i.e. straight-line pieces of code without any jumps nor system or function calls.*
- *Finally,*
  $$\delta_P \subseteq (\mathcal{S} \times \mathcal{G} \times FuncC \times \mathcal{S}) \cup (\mathcal{S} \times \mathcal{G} \times SysC \times \mathcal{S})$$
  is a transition relation. □

Intuitively, these action guarded transitions should satisfy the following conditions:

- If there is a function call $F \in FuncC$ at a program point $v \in V$ then $\exists s, \exists s' \in \mathcal{S}$ and $\exists g \in \mathcal{G}$ s.t. $v \in \lambda(s)$, and $\delta_P(s,g) = (f, s')$. Also the predicate $P(F,s)$ is in $Pred_{site}$.
- If there is a system call $C \in SysC$ at a program point $v \in V$ then $\exists s, \exists s' \in \mathcal{S}$ and $\exists g \in \mathcal{G}$ s.t. $v \in \lambda(s)$, and $\delta_P(s,g) = (c, a, s')$.

As we want to monitor only system calls, we need to replace any function calls to a function F by the entry of the submodel of F. But we need to keep the control location to which the application should return after reaching the exit state of submodel of F.

Model 3. ***(GSCG: Guarded System Call Graph)*** *Let P be a procedure and let guarded call graph (GCG) be*

$$G = \langle \mathcal{S}, Enter_P, Exit_P, \delta_{SC}, \lambda, FuncC, SysC, Pred_{site} \rangle.$$

Then G is said to be a Guarded System Call Graph *if $FuncC = \emptyset$. In other words, G does not generates function calls. Furthermore, the transition relation δ_{SC} is defined as*

$$\delta_{SC} \subseteq (\mathcal{S} \times \mathcal{G} \times SysC \times (A \cup \{epsilon\}) \times \mathcal{S})$$

where the finite set of actions A is given by the grammar

$$a \leftarrow P := \mathtt{True} | P := \mathtt{False} | \epsilon,$$

where P is a predicate in $Pred_{site}$. □

Now the transition relation only considers system calls. But, again, it is very important to keep the information related to function call sites and return sites. In order to do so, we complete the transition relation with assignments of the form $P(s, F) :=$ `True`, or $P(s, F) := 1$, that indicate that a function F has been called at control point s (the call was done at a location v such that $\lambda(v) = s$). In order to indicate a return from a call to function F, we use the assignment of the form $P(s, F) :=$ `False`, or $P(s, F) := 0$, where $P(s, F)$ is a predicate in $Pred_{site}$. In order to be able to handle recursive functions, we extend the GSCG model to a PDS, i.e., we add a stack.

Model 4. ***E_GSCG: Extended GSCG*** *Let P be a procedure and let its guarded call graph (GSCG) be*

$$G = \langle \mathcal{S}, Enter_P, Exit_P, \delta_P, \lambda, FuncC, SysC, Pred_{site} \rangle.$$

The E_GSCG E_G extends G with a finite stack alphabet

$$\Gamma \subseteq \{Pred_{site} \times \{0, 1\}\},$$

where $\gamma_0 \in \Gamma^*$ the initial stack configuration. We define the transition system δ_E as follow:

- *A_E is the finite set of actions given by the following grammar:*

  $$a \leftarrow \mathsf{Push}(\gamma) | \mathsf{Pop}(\gamma) | B := 0 | B := 1 | \epsilon,$$

  where $B \in Pred_{site}$, $\gamma \in \Gamma$ and $\mathsf{Push}(\gamma), \mathsf{Pop}(\gamma)$ are the two common operations that pop and push γ into the stack.

- *Finally,*

  $$\delta_E \subseteq \cup(\mathcal{S} \times \Gamma \times \mathcal{G} \times SysC \times (A_E \cup \{\epsilon\}) \times \mathcal{S} \times \Gamma).$$

□

We present the semantic of its transition relation:

Definition 6.2. ***Run of an E_GSCG.***
For sequence of system calls, i.e. a word

$$w = a_1, ..., a_k \in Syc_{call}$$

a run of G_P on w is a sequence

$$\rho = (s_0 \rho_0), \ldots, (s_k, \rho_k),$$

where for every $s_i \in \mathcal{S}, \rho_i \in \Gamma^*$, $\rho_0 = \epsilon$ and for every $1 \leq i \leq k$ the following rules apply:

- Making a system call*:*
  If $(s_i, \epsilon, g, a_i, \epsilon, s_{i+1}, \epsilon) \in \delta$ *then the system call* a_i *is allowed when transitioning from* s_i *to* s_{i+1} *if the guard* $g \in \mathcal{G}$ *evaluates to* True *at* s_i *and there is no stack activity* $(\rho_{i+1} = \rho_i)$*.*

- Call to a non-recursive function*:*
  If $(s_i, \epsilon, g, a_i, (P(s_i, F) := \texttt{True}), s_{i+1}, \epsilon) \in \delta_E$ *the system call* a_i *is allowed when transitioning from* s_i *to* s_{i+1} *if the guard* $g \in \mathcal{G}$ *is evaluate to* True *at* s_i*. The action of the transition sets the predicate* $P(s_i, F) \in Pred_{site}$ *to* True*, meaning that a call to a function* F *has been done at* s_i *and* a_i *is the first system call induced by* F *for the current context. Also, there is no stack activity.*

- Return from a non-recursive function*:*
  If $(s_i, \epsilon, (P(s_i, F) == \texttt{True}) \wedge g), a_i, (P(s_{i+1}, F) := \texttt{True}), s_{i+1}, \epsilon) \in \delta_E$ *then the system call* a_i *is* allowed *by our model when transitioning from* s_i *to* s_{i+1} *if the guard* $(P(s_i, F) == \texttt{True}) \wedge g$ *evaluates to* True *at* s_i*. The action of the transition assigns the value of the predicate* $P(s_{i+1}, F) \in Pred_{site}$ *to* False*, meaning that* s_{i+1} *is the return address of the non-recursive function* F*. In the case where there are many call sites to* F*, the guard checks also if it is the correct return address* $(P(s_i, F) == \texttt{True})$*. Also, there is no stack activity.*

- Call to a recursive function*:*
  If $(s_i, \epsilon, g, a_i, \textsf{Push}(\gamma), s_{i+1}, \gamma) \in \delta_E$ *then the system call* a_i *is allowed by our model when transitioning from* s_i *to* s_{i+1} *if the guard* g *evaluates to* True *at* s_i*. In that case,* γ *is* pushed *onto the predicate stacks, meaning that* $\rho_{i+1} = \gamma \cdot \rho_i$*, with* s_i *interpreting the function call site mentioned in* γ*, and where "$\cdot$" is the operator of concatenation.*

- Return from a recursive function*:*
  If $(s_i, \gamma, (P(s_{i+1}, F) == \texttt{True}) \wedge g, a_i, \textsf{Pop}(\gamma), s_{i+1}, \epsilon) \in \delta_E$ *and* $\gamma = (P(s_{i+1}, F), 1)$*. The system call is allowed if* g *and the test at the* $\textsf{Top}$ *of the predicate stack* $(P(s_{i+1}, F) == \texttt{True})$ *are* True*. It means that the program returns from a function call and pops* γ *from the stack under the condition that* γ *is at the top of the stack.* □

Monitor using Pushdown System against stream of system calls are in fact *visibly pushdown automaton* [81]. In these models, the stack alphabet is defined by a set of return addresses referring to function call returns locations. At function call cites, it pushes onto the stack the assocated address. When the application return, it pops the address at the top of the stack only when returning. There is no stack activity otherwise. Unfortunately, these visibly pushdown systems requires the introduction of extra system calls at each function call locations into the original application that we which to protect. Our model do not requires such expensive code instrumentations as we use the predicates in $Pred_{site}$ and a *stack of predicates* controlling the return of a function called at the correct address. Our monitor reaches stack determinism. Recent H-IDS uses Dyck model [75] to monitor the application. But these models requires codes instrumentations inserting extra system calls at each function call cites in the application. These code rewrittings imply very important overheads at the monitoring. Also, such models do not handels recusive calls. The models proposed in [72] requires the monitoring of program counter. Our model handles recursive call, does not add extra system calls and does not needs to monitor the program counter. Our models can be simulated by a Dyck model [75]. This statement insure that our model detects more attacks than a Dyck model [75].

Theorem 6.1. *Let $\mathcal{P}$ be a program. g*

- *If $\mathcal{P}$ has no recursive functions then its GSCG can be simulated by a Dyck Model.*
- *Otherwise, if $\mathcal{P}$ contains recursive functions then its E_GSCG can be simulated by a Dyck Model.* □

Proof. Let G be the GSCG of $\mathcal{P}$ and $t \in \delta_P$ such that $t = (s, g, C, A, s')$.

- If the action A is of the form $Pred(s, F) := \mathsf{True}$ then there exist a transtion in the Dyke Model departing from s and labeled by Pre_Call_F.
- If the action is of the form $Pred(s, F) := \mathsf{False}$ and the guard $Pred(s, F) == \mathsf{True}$ then there exists a transition departing from s labeled by $Post\_Call\_F$ and the top of the Dyke stack is Pre_Call_F.

Thus, we proved that the GSCG is simulated by a Dyck Model in the case where there is no recursive calls.
Now, let G be the E_GSCG of $\mathcal{P}$ and $t \in \delta_E$.

- If $t = (s, \epsilon, g, C, \mathsf{Push}(\gamma), s', \gamma)$ and $\gamma = (Pred(s, F), 1)$. Then there exist a transtion in the Dyke Model departing from s and labeled by Pre_Call_F. At s' the top of the Dyke stack is Pre_Call_F and the top of E_G's stack is $(Pred(s, F), 1)$.
- If $t = (s, \gamma, (Pred(s, F) == \mathsf{True}) \wedge g, C, \mathsf{Pop}(\gamma), s', \epsilon$ and $\gamma = ((Pred(s', F), 1)$. Then there exists a transition departing from s labeled by $Post\_Call\_F$ and the top of the Dyke stack is Pre_Call_F.
- The other cases are the same as these describe for the $GSCG$.

Thus we proved that in presence of recursive calls, E_GSCG can be simulated by Dyke Model. For all sequences of system calls $w \in SysC^*$ generated by the *possible* execution of $\mathcal{P}$, if there exists an accepted run of G on w then there exists the corresponding run in the Dyke Model. □

The following example illustrate the vulnerability of the state-of-the-art models for H-IDS. It also shows how we insure control and data flow integrity while limiting dramatically the attacker's ability to execute mimicry or non-control-data flow attacks.

Example 6. *Consider the following program:*

```
1  ...{
2  char *str, *user; int j;
3  ...
4  if (strncmp (user,"admin",5)==0){
5    sys_1();}
6  else {
7    sys_2();}
8  user = "gest";
9  strcpy (str, someinput); /* <- Exploit! */
10 if (strncmp (user,"admin",5)==0){
11   sys_3();}
12 else {
13   sys_4();}
14 j = F(user);
15 if (j>0){
16   sys_7();}
17 else {
18   sys_8();}
19 ...}
20 int F(char *user1) {
21 if (strncmp (user1,"admin",5)==0){
22   sys_5();}
23 else {
24   sys_6();}
25 return 0;
26 }
```

First, we illustrate our model. In this case it can be seen as an abstract state graph of the code. We recall that our induced GSCG Model (see Model 3) in our momitor is build using predicate abstraction. Here, we use the predicates

$$B_1 \equiv (user = admin) \text{ and } B_2 \equiv (j > 0),$$

present in the conditional branches.

$$\begin{tikzcd}
\boxed{l_4} \arrow[r, bend left, "\frac{\neg B_1}{sys\_2(\ );}"] \arrow[r, bend right, "\frac{B_1}{sys\_1(\ );}"'] & \boxed{l_{10}} \arrow[r, "\frac{\neg B_1}{sys\_4(\ );}"'] & \boxed{l_{F21}} \arrow[r, "\frac{\neg B_1}{sys\_6(\ );}"'] & \boxed{l_{15}} \arrow[r, "\frac{\neg B_1 \wedge \neg B_2}{sys\_8(\ );}"'] & \boxed{l_{19}}
\end{tikzcd}$$

If we consider the program made of the code lines from 1 *to* 7 *and from* 9 *to* 13 *only we found the example presented in [73] Section "7. Limitations" that was used in order to illustrate the limitation of their Dyke Models. As the authors of [73] noticed, considering only these piece of code lines, the Dyke Model would allow the following four sequences of system calls in a nondeterministic way:*

```
[[sys_1,sys_3],[sys_1,sys_4],[sys_2,sys_3],[sys_2,sys_4]]
```

Considering still the code depectied in lines $1 - -7$ *and* $9 - -13$ *our monitor allows deterministically only the following two system calls sequences that are actually the only one that the program can generate in a safe environment:*

```
[[sys_1,sys_3],[sys_2,sys_4]]
```

Now, let us consider the complete program depicted above. The complete program would allow us to express and illustrate deeper the comparaison between our model and the State-of-the-art H-IDS using Dyke-based model (e.g., [75], [73]). First, we provide below the list of all 16 *streams of system calls that are allowed by a H-IDS using a Dyke like model as monitor for this program:*

```
[
[sys_1,sys_3,sys_5,sys_7],
[sys_1,sys_3,sys_5,sys_8],
[sys_1,sys_3,sys_6,sys_7],
[sys_1,sys_3,sys_6,sys_8],
[sys_1,sys_4,sys_5,sys_7],
[sys_1,sys_4,sys_5,sys_8],
[sys_1,sys_4,sys_6,sys_7],
[sys_1,sys_4,sys_6,sys_8],
[sys_2,sys_3,sys_5,sys_7],
[sys_2,sys_3,sys_5,sys_8],
[sys_2,sys_3,sys_6,sys_7],
[sys_2,sys_3,sys_6,sys_8],
[sys_2,sys_4,sys_5,sys_7],
[sys_2,sys_4,sys_5,sys_8],
[sys_2,sys_4,sys_6,sys_7],
[sys_2,sys_4,sys_6,sys_8],
 ]
```

Considering this same complete program, our model allows only the following 2 *streams of system calls:*

```
[[sys_1,sys_4,sys_6,sys_8],[sys_2,sys_4,sys_6,sys_8]]
```

In fact it is the only 2 *streams of system calls possibles for the program running in a safe environment. Our model will raise deviation report on any other sequences of system calls. The other* 14 *sequences of system calls that the Dyke Model allows are actually real violations and deviations of a normal execution of the code. Now, line xx can be seen as a very useful exploit for an attacker that could for instance overflow* str *with a very large value for* someinput *in order to rewrite the memory and change the value of* user *to "*admin*". In the case where the H-IDS uses a Dyke like model, the attacker will be able to navigate and execute any sequences of system calls without being detected. In the case where our model is in use, the attacker will be detected as soon as he try to execute one of the abnormal sequences. Thanks to the abstract call graph embeeded in our monitor the predicate* $\neg B_1 \equiv \neg(user = admin)$ *holds at any system calls cites after the location of the exploit (line xx).* □

In terms of effectiveness, we could mathematically prove that the automatically generated mimicry attacks from [78] will inevitably violate one of our pre-generated invariants. In terms of precision propoerties for our model, we note that if our generated abstract graph remains deterministic, we note that it is an exact abstraction of the real codes. Thus, we can use this model directly to check for logic bugs and vulnerabilities in the application. As we can see, the theoretical framework introduced in the previous sections has been established. Moreover, we note that any other assertion checker, invariant generation or dynamic tools can be considered in this framework.

7 Conclusion

We provide an extensible invariant-based formal platform for malware analysis. These invariants concisely capture the semantic of the malicious behavior. Our platform is flexible as any invariant generation method could be incorporated. In other words, it is an open architecture, where any invariant generation tool can be connected into a *tool bus* where invariants, expressed in different logics, will help in the identification of malicious behavior or in the construction of a precision intrusion detection system.

We provide formal malware signatures generation methods and formal program models. Considering each formal specification techniques and Models we were able to reduce the malware detection problem to classical problems addressed by the formal methods community. We identify and express exactly the obtained formal problem. And we adapted existing formal approaches in order to build new theoretical fundations for malware static analysis platform design. Following the lines of such theoretical frameworks, we proposed host-based intrusion detection systems.

References

[1] Nachenberg, C.: Computer virus-antivirus co-evolution. Commun. ACM **40**(1) (1997) 46–51

[2] Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Computer Aided Verification. Volume 1254 of LNCS., Haifa, Israel, springer (June 1997) 72–83

[3] S. Bensalem, Y. Lakhnech, H. Saidi: Powerful techniques for the automatic generation of invariants. In Rajeev Alur, Thomas A. Henzinger, eds.: Proc. of the 8th Int. Conf. on Computer Aided Verification CAV. Volume 1102., NJ, USA (1996) 323–335

[4] Tiwari, A., Ruess, H., Saidi, H., Shankar, N.: A technique for invariant generation. In: TACAS: Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. (2001)

[5] Rebiha, R., Moura, A.V., Matringe, N.: Generating invariants for non-linear loops by linear algebraic methods. Technical Report IC-13-04, Institute of Computing, University of Campinas (February 2013)

[6] Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In Sestoft, P., ed.: European Symp. on Programming, ESOP 2006. Volume 3924 of LNCS. (2006) 279–293

[7] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Proceedings of the 21st International Conference on Computer Aided Verification. CAV '09, Berlin, Heidelberg, Springer-Verlag (2009) 661–667

[8] Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings. (2004) 53–68

[9] Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: In CAV. (2009)

[10] Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (jun 1992) 748–752

[11] Paulson, L.C.: Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow). Volume 828 of Lecture Notes in Computer Science. Springer (1994)

[12] The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2004) Version 8.0.

[13] Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2010)

[14] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In Gupta, A., Kroening, D., eds.: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK). (2010)

[15] Barrett, C., Tinelli, C.: CVC3. In Damm, W., Hermanns, H., eds.: Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). Volume 4590 of Lecture Notes in Computer Science., Springer-Verlag (July 2007) 298–302 Berlin, Germany.

[16] Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proceedings of the 19th International Conference on Computer Aided Verification. CAV'07, Berlin, Heidelberg, Springer-Verlag (2007) 519–531

[17] Dutertre, B., Moura, L.D.: The yices smt solver. Technical report (2006)

[18] Gulwani, S., Tiwari, A.: Assertion checking unified. In: Sixth Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05). (2005)

[19] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In Ball, T., Rajamani, S.K., eds.: Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings. Volume 2648 of Lecture Notes in Computer Science., Springer (2003) 235–239

[20] Xie, Y., Aiken, A.: Saturn: A sat-based tool for bug detection. (2005) 139–143

[21] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrÉe analyzer. In: Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science, Springer (2005) 21–30

[22] Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-soft: Software verification platform. In: Proceedings of the 17th International Conference on Computer Aided Verification. CAV'05, Berlin, Heidelberg, Springer-Verlag (2005) 301–306

[23] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper., Hyderabad, India (December 2008)

[24] : BitBlaze: Binary analysis for computer security `http://bitblaze.cs.berkeley.edu/`.

[25] Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Proceedings of the 5th Colloquium on International Symposium on Programming, London, UK, Springer-Verlag (1982) 337–351

[26] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge, MA (2000)

[27] McMillan, K.L.: Symbolic Model Checking. Kluwer Qcqdemic Publishers, Norwell, MA, USA (1993)

[28] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8) (1986) 677–691

[29] Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. **58** (2003)

[30] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, ACM Press, NY (1977) 238–252

[31] Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming **13**(2–3) (1992) 103–179

[32] Colon, M.A., Uribe, T.E., Col'on, M.A., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: in Computer Aided Verification, Springer (1998) 293–304

[33] Saidi, H., Shankar, N.: Abstract and model check while you prove. In: CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (1999) 443–454

[34] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (2003) 752–794

[35] Saidi, H.: Modular and incremental analysis of concurrent software systems. In: ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering, Washington, DC, USA, IEEE Computer Society (1999) 92

[36] Dutertre, B., Moura, L.D.: A fast linear-arithmetic solver for dpll(t), Springer (2006) 81–94

[37] Baelde, D., Gacek, A., Miller, D., Nadathur, G., Tiu, A.: The bedwyr system for model checking over syntactic expressions. In: CADE-21: Proceedings of the 21st international conference on Automated Deduction, Berlin, Heidelberg, Springer-Verlag (2007) 391–397

[38] Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer-Verlag New York, Inc., New York, NY, USA (1995)

[39] Dijkstra, E.W.: A Discipline of Programming. Prentince-Hall (1976)

[40] Floyd, R.W.: Assigning meanings to programs. In: Proc. 19th Symp.Applied Mathematics. (1967) 19–37

[41] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (1969) 576–580

[42] Browne, I.A., Manna, Z., Sipma, H.B., Manna, Z., Sipma, H.B.: Generalized temporal verification diagrams. In: In 15th Conference on the Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag (1995) 726–765

[43] Katz, S., Manna, Z.: A heuristic approach to program verification. In: In the Third International Joint Conference on Artificial Intelligence, Stanford, CA (1973) 500–512

[44] German, S.M.: A program verifier that generates inductive assertions. In: Technical Report TR, Center for Research in Computing Technology, Harvard U. (1974) 19–74

[45] Wegbreit, B.: The synthesis of loop predicates. Commun. ACM **17**(2) (1974) 102–113

[46] Karr, M.: Affine relationships among variables of a program. Acta Inf. **6** (1976) 133–151

[47] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Tucson, Arizona, ACM Press, New York, NY (1978) 84–97

[48] Bjorner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theor. Comput. Sci. **173**(1) (1997) 49–87

[49] McGraw, G., Morrisett, G.: Attacking malicious code: A report to the infosec research council. IEEE Software **17**(5) (2000) 33–41

[50] Adelman, L.M.: An abstract theory of computer viruses. In: Advances in Cryptology CRYPTO'88. (1988)

[51] Cohen, F.: A short courses on computer viruses. In: Wiley,. (1990)

[52] Fisher, C.: Tremor analysis(pc). In: Virus Diget, 6(88). (1993)

[53] Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, Department of Computer Science, Auckland New Zealand (1997)

[54] Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005), Oakland, CA, USA, ACM Press (May 2005) 32–46

[55] Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). (2007)

[56] Dalla Preda, M., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. In: POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2007) 377–388

[57] Baader, F., Siekmann, J.H.: Unification theory. (1994) 41–125

[58] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002) 213–228

[59] : APRON: numerical anstract domain library `http://apron.cri.ensmp.fr/library/`.

[60] : The interproc analyzer http://pop-art.inrialpes.fr/interproc/interprocweb.cgi.

[61] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: Applications to software engineering. In: Journal on Software Tools for Technology Transfer (paper from FASE2005). (2007)

[62] Henzinger, T.A., Hottelier, T., Kovács, L.: Valigator: A verification tool with bound and invariant generation. In: In Proc. of LPAR. (2008) 333–342

[63] Hoder, K., Kovacs, L., Voronkov, A.: Invariant generation in vampire. In: Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software. TACAS'11/ETAPS'11, Berlin, Heidelberg, Springer-Verlag (2011) 60–64

[64] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1–3) (December 2007) 35–45

[65] Dijkstra, E.W.: A Discipline of Programming. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)

[66] Gopan, D., Reps, T.W.: Low-level library analysis and summarization. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer (2007) 68–81

[67] Brumley, D., Jager, I.: The bap hanbook. In: Technical Report TR. (2009)

[68] Song, F., Touili, T.: Pushdown model checking for malware detection. In: Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. (2012) 110–125

[69] Emerson, E.A.: Handbook of theoretical computer science (vol. b). MIT Press, Cambridge, MA, USA (1990) 995–1072

[70] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2) (April 1986) 244–263

[71] Dutertre, B.: Yices 2.2. In Biere, A., Bloem, R., eds.: Computer-Aided Verification (CAV'2014). Volume 8559 of Lecture Notes in Computer Science., Springer (July 2014) 737–744

[72] Wagner, D., Dean, D.: Intrusion detection via static analysis. In: IEEE Symposium on Security and Privacy. (2001) 156–169

[73] Feng, H.H., Giffin, J.T., Huang, Y., Jha, S., Lee, W., Miller, B.P.: Formalizing sensitivity in static analysis for intrusion detection. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2004) 194

[74] Gopalakrishna, R., Spafford, E.H., Vitek, J.: Efficient intrusion detection using automaton inlining. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2005) 18–31

[75] Giffin, J.T., Dagon, D., Jha, S., Lee, W., Miller, B.P.: Environment-sensitive intrusion detection. In Valdes, A., Zamboni, D., eds.: Recent Advances in Intrusion Detection (RAID). Volume 3858 of Lecture Notes in Computer Science., Springer (2005) 185–206

[76] Forrest, S., Longstaff, T.: A sense of self for unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy. (May 1996) 120–128

[77] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In Sandhu, R., ed.: Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, DC, USA, ACM Press (November 2002)

[78] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: Proceedings of the 14th USENIX Security Symposium. (2005)

[79] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., nkar K. Iyer, R.: Non-control-data attacks are realistic threats. In: USENIX Security Symposium. (2005)

[80] Rebiha, R., Saidi, H.: Quasi-static binary analysis: Guarded model for intrusion detection. In: TR-USI-SRI-11-2006. (November 2006)

[81] Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, New York, NY, USA, ACM (2004) 202–211