SMT Solvers for Software Security

Julien Vanegue
jvanegue@microsoft.com
Microsoft Security Science

Sean Heelan
sean.heelan@gmail.com
Immunity Inc.

Rolf Rolles
rolf.rolles@gmail.com
Unaffiliated

Abstract

Computational capacity of modern hardware and algorithmic advances have allowed SAT solving to become a tractable technique for the resolution of decision problems derived from complex software. In this article, we present three practical applications of SAT to software security in static vulnerability checking, exploit generation, and the study of copy protections. These areas are some of the most active in terms of both theoretical research and practical solutions. Investigating the successes and failures of approaches to these problems is instructive in providing guidance for future work on the problems themselves as well as other SMT-based systems.

1 Introduction

Satisfiability (SAT) is a decidable computational problem with the following structure: given a boolean formula, is there a valuation of the variables for which the formula is true? If such a valuation exists, the formula is said to be *satisfiable*. If there is no such valuation, the formula is said to be *unsatisfiable*. The complexity of SAT is *NP complete* which means that there is no efficient algorithm to resolve all instances of the problem. Therefore, a variety of heuristics are used to lower the execution time of decision procedures in practice. Yet, the SAT problem is the object of active research as more optimized strategies are crafted and larger scale experiments with conclusive results are performed every year. More recently SAT solvers have become the driving engines behind a more expressive approach to constraint specification and solving. The *Satisfiability Modulo Theories* (SMT) problem extends the SAT problem with support for higher level theories, such as bitvector arithmetic among others, and relational operators, such as equality. Due to these higher level concepts, SMT allows for a more natural modeling of the semantics of code and as such is typically used instead of SAT in reasoning about properties of software applications. In this paper, we study the practical use of SMT solvers as *black box oracles* to answer questions that encode the essence of problems of static vulnerability checking, exploit generation, and analysis of copy protection. We show that SMT solvers are convenient tools to decide many important security queries about programs. Nonetheless, a key point throughout is the separation of concerns between constraint generation and constraint solving. A solver is not in charge of the constraint generation step but a failure to generate constraints that accurately model properties of the system under inspection limits the relevance of the answers a solver can provide. The constraint generation step is known as the *inference* problem. Inference consists of automatically generating constraints from inspection of the analyzed program. Those constraints are passed to the SMT solver for resolution. Constraint generation and solving been studied in functional programming and compilers [28] but less so in applications related to software security [9]. While one might expect resolution to be the bottleneck in systems based on SMT solvers, we will later discuss how, in many instances, the problem of constraint generation is currently the main limiting factor.

1.1 Vulnerability checking

The last decade has seen some success in deploying program verification tools to industrial software. The main techniques for program verification are theorem proving [3], abstract interpretation [12] and model checking [11]. In this article, we focus on security vulnerability checking based on theorem proving for imperative programs written in C language using the HAVOC [21] tool. Theorem proving is a mature technique that has been applied to verify call-free, loop-free programs and

large hardware systems [22]. Applications of theorem proving to software is more recent thanks to global analysis techniques such as *Predicate abstraction* [4]. Predicate Abstraction is a potentially diverging but automated technique to infer constraints at procedure and loop boundaries. Software model-checking tools based on predicate abstraction (such as SLAM [4]) have brought considerable value for automated analysis of software by uncovering hundreds of software bugs in medium-sized drivers. To address the scalability issues arising from the state space explosion problem induced by model checking, the Houdini algorithm [17] has been devised to answer the problem of monomial predicate abstraction. Houdini is a simple yet powerful technique based on *candidate contracts* (or *may be constraints*) allowing the user to provide simple constraints templates and using the constraint solver in a fixed-point algorithm to determine whether or not those constraints always hold at function or loop boundaries. Houdini is implemented in the Boogie verification framework [24] for which HAVOC is a front-end. While Houdini is a terminating and deterministic algorithm, it is unable to answer existential queries, as candidate constraints are only persisted when they hold in every function contexts. Thus, Houdini cannot be used to answer the following question: *is it feasible for parameter p to hold value v in some context?* . On the other hand, it can answer questions such as: *is it provable that parameter p always has value v?*.

Section 2 illustrates the inference problem on C programs using a simple yet non-trivial loop program based on a Sendmail vulnerability [33] for which the SMT solver does an excellent job at deciding satisfiability of a set of constraints at given program points, but does not provide a mechanism to synthesize the required constraints automatically. While Houdini is able to reason about candidate loop invariants, automatically inferring such complex invariants is out of reach. As such, the constraints fed to the solver are often provided by an expert analyst or generated using limited strategies.

1.2 Exploit generation

Exploit generation is a more recent field of study than vulnerability checking. Work so far has primarily fallen into two categories — attempts at automated generation of inputs aimed at hijacking the control flow of a system [6, 20, 1] and attempts at automating the generation of malicious payloads [29, 34, 16, 32]. Work in the former category has relied on symbolic/concolic execution systems [35, 7, 10, 5] to perform constraint generation. Such systems track the semantic relationship and constraints between symbolic input data and all other bytes in program memory and CPU registers. Using this information it is then possible to generate SMT formulae that ask questions about the potential values of such bytes. Exploit generation systems have utilized this ability to check if data that correspond to potentially sensitive pointers can be controlled by an attacker. So far the problems tackled in this area have been simplified scenarios where commonplace operating system security measures and binary hardening techniques are disabled. The reasons these limitations have been necessary will be discussed in section 3.1.

The research performed on generation of malicious payloads has had more success at solving real world problem instances. For the purposes of this article we define the *payload* of an exploit to be code executed once the control flow of an application has been hijacked. The payloads generated as part of exploitation research have so far been sequences of *return-oriented programming* (ROP) gadgets. A gadget is a sequence of instructions, within a shared library or executable in the target program, that performs some useful computation and ends by transferring control flow to the next gadget in the sequence. A collection of such gadgets is usually chained together to accomplish a specific task, such as changing permissions of a memory segment, copying in a second stage payload and then executing that second stage. This approach to executing malicious code is necessary to deal with protection mechanisms that prevent one from executing code that has been placed on the stack or heap. By instead executing sequences of instructions within the program's code this mechanism can be avoided and sometimes disabled entirely.

SMT solvers have been used as the reasoning component of systems that prove functional equivalence between a desired computation and a sequence of instructions [34]. They have also formed part of end-to-end ROP compilers [16, 32] that attempt to automatically chain together sequences of such gadgets so that the sequence is semantically equivalent to a model payload. These systems usually incorporate a SMT solver as a small part of an larger set of algorithms. Section 3.2 discusses both approaches and their integration of SMT solvers. As with many other successful applications of SMT solvers, there is a focus on reducing the number of queries that must be made and pre-processing the input to a solver through other algorithms in order to reduce the complexity of each query.

1.3 Copy protection analysis

In the domain of copy protections, we consider the two problems of equivalence checking of obfuscated programs and automated cryptanalysis. We find, similarly

to the aforementioned domains of vulnerability checking and exploit generation, that the main issue that is faced is not so much feeding a constraint system into an SMT solver, but rather, how to generate the constraint system from the program (and which constraint system to generate) in the first place. For our application of equivalence checking to verify the proper working of a deobfuscator for a virtualization obfuscator, the use of an SMT solver is easily feasible. If we wish to apply the same methodology to simply verifying that a virtualized program is equivalent to its original version, we run into issues surrounding input-dependent branches in the obfuscated version. For our application of automated cryptanalysis, problems exist surrounding how properly to model code with input-dependent branches, particularly input-dependent loops over unbounded quantities. Other program-analytic techniques such as invariant inference can furnish solutions to these problems, but they fall under the inference phase and are orthogonal to the actual solving of the system.

2 SMT in vulnerability checking

In this section, we use a verification tool HAVOC [21] (a heap-aware verifier for C and C++ programs) to translate to an intermediate form Boogie [24] which then calls SMT solver Z3 [26] to decide a vulnerable program from a non-vulnerable one. For conciseness, we do not make explicit the step of transforming the source code into an intermediate representation (IR) and go straight to the construction of the formula. The Boogie IR is based on the Static Single Assignment form (SSA [13]) which makes it easier to construct the final formula passed to the solver. We use the code snippet [15] in Figure 1, a simplified version of the Sendmail crackaddr vulnerability [33] published by Mark Dowd in 2003. This example contains a non-trivial loop program parsing an untrusted string parameter.

```
 1: #define BUFFERSIZE 25
 2:
 3: int copy_it(char *input)
 4: {
 5:  char lbuf[BUFFERSIZE];
 6:  char c, *p = input, *d = &lbuf[0];
 7:  char *ulimit = &lbuf[BUFFERSIZE-10];
 8:  int quotation = FALSE;
 9:  int rquote = FALSE;
10:
11:  memset(lbuf, 0, BUFFERSIZE);
12:
13:  while((c = *p++) != '\0')
14:  {
15:    if ((c == '<') && (!quotation))
16:    {
17:      quotation = TRUE;
18:      ulimit--;
19:    }
20:    if ((c == '>') && (quotation))
21:    {
22:      quotation = FALSE;
23:      ulimit++;
24:    }
25:    if (c == '(' && !quotation && !rquote)
26:    {
27:      rquote = TRUE;
28:      // FIX: insert ulimit--; here
29:    }
30:    if (c == ')' && !quotation && rquote)
31:    {
32:      rquote = FALSE;
33:      ulimit++;
34:    }
35:    if (d < ulimit)
36:      *d++ = c;
37:  }
38:  if (rquote)
39:    *d++ = ')';
40:  if (quotation)
41:    *d++ = '>';
42: }
```

Figure 1: Essence of the Sendmail crackaddr vuln.

A buffer overflow vulnerability exists at line 36 due to a missing decrement of the *ulimit* variable. The correct fix for this loop is to enable such decrement at line 28. At first sight, this loop seems rather complex to verify. A few key remarks about the structure of the loop are fundamental in understanding the behavior of this function. First, the function contains two state variables *quotation* and *rquote* corresponding to the value of the currently processed character as pointed by variable *c* . The two state variables hold value *False* at the initial state of the loop. Only a small number of combinations of state values are possible due to the fact that the inner conditionals in the loop are mutually exclusive, since the value of variable *c* does not change within the same iteration of the loop. The second fundamental remark on this loop is about the *ulimit* variable. The initial value of this variable as assigned on line 7 is pointing on offset 15 of the local buffer (since BUFFERSIZE equals 25). Depending on the processed input string, this value can either be incremented or decremented. It is possible to model the expected behavior of the loop using a finite state automata corresponding to the expected and valid results of its computations (with line 28 enabled). Here, we model the state of the loop as a triple of type *(bool,bool,int)* corresponding to the value of variables *quotation*, *rquote* and the value of *upperlimit - lbuf* as shown in Figure 1.

Note that *upperlimit* is a pointer variable and not an integer offset, thus the numerical value in the third component of the triple should in fact be read *lbuf* +

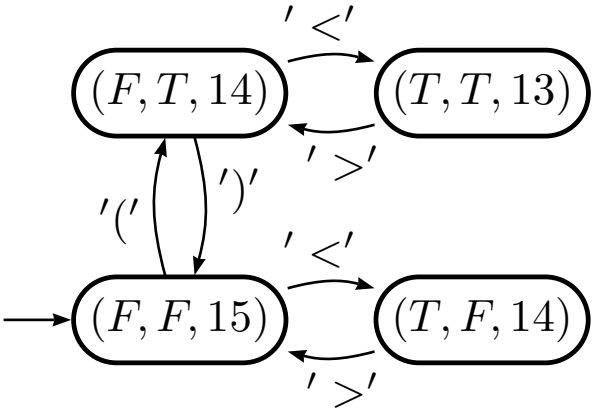

Figure 2: Automaton corresponding to loop in Fig. 1

num where *num* is the relative offset from the beginning of the buffer. We represent this value by only displaying *num* in the automaton for conciseness. The absence of line 28 in the studied loop introduces a problematic transition in this automata, since the *ulimit* variable is not bounded anymore. It is then possible to assign a value to *ulimit* that is big enough to trigger an out of bound write access to the local buffer on line 35. We can formalize a logical representation corresponding to this automata by ignoring transitions and only retaining state values. The automaton then corresponds to the following formula P :

$$\begin{aligned}&(ulim = lbuf + 15 \wedge quotation = F \wedge rquote = F)\\ \vee\ &(ulim = lbuf + 14 \wedge quotation = T \wedge rquote = F)\\ \vee\ &(ulim = lbuf + 14 \wedge quotation = F \wedge rquote = T)\\ \vee\ &(ulim = lbuf + 13 \wedge quotation = T \wedge rquote = T)\end{aligned}$$

The technique used by SMT solvers to verify such invariant is known as *proof by induction*. In a nutshell, a proof by induction involves two steps:

1. Prove that the formula holds for the base case (at the entry point of the loop) : $P(0)$
2. Prove that if the formula holds at iteration *n* of the loop, then it also holds at the next iteration : $\forall n : P(n) \Rightarrow P(n+1)$

$P(0)$ means invariant P holds at the entry state while P(n) means that P holds at the n^{th} iteration. Such formula is indeed inductive and easily solved by a SMT solver. We now give the full version of this proof.

Proof. The proof involves analysis of all possible loop states and valid transitions between states. It is trivial to notice that the formula indeed holds at the entry point of the loop since the entry state (ulimit = lbuf + 15 $\wedge$ quotation = F $\wedge$ rquote = F) exactly corresponds to the value of the variables at the loop entry. Thus $P(0)$ holds.

The next step consists of the following: *assuming* that the loop is in one of the states described by the invariant at the beginning of the iteration, does the loop remain in a state described by the invariant? There are four cases to consider (assuming the loop start in one of the four states) and four sub-cases for each case (assuming we take one of the four available transition, corresponding to one of the four conditionals of the loop). In practice, not all transitions are available from all states. As such, the proof will be smaller than unrolling 16 different cases.

1. Assume that the loop iteration starts in state (ulimit = lbuf + 15 $\wedge$ quotation = F $\wedge$ rquote = F).
   (a) The loop enters state (ulimit = lbuf + 14 $\wedge$ quotation = T $\wedge$ rquote = F) if it executes the first conditional (lines 15-19)
   (b) The loop enters state (ulimit = lbuf + 14 $\wedge$ quotation = F $\wedge$ rquote = T) if it executes the third conditional (lines 25-29)
   (c) No other conditional can be entered from such entry state.
2. Assume that the loop iteration starts in state (ulimit = lbuf + 14 $\wedge$ quotation = T $\wedge$ rquote = F).
   (a) The loop enters state (ulimit = lbuf + 15 $\wedge$ quotation = F $\wedge$ rquote = F) if it executes the second conditional (lines 20-24)
   (b) No other conditional can be entered from such entry state.
3. Assume that the loop iteration starts in state (ulimit = lbuf + 14 $\wedge$ quotation = F $\wedge$ rquote = T).
   (a) The loop enters state (ulimit = lbuf + 15 $\wedge$ quotation = F $\wedge$ rquote = F) if it executes the fourth conditional (lines 30-34)
   (b) The loop enters state (ulimit = lbuf + 13 $\wedge$ quotation = T $\wedge$ rquote = T) if it executes the first conditional (lines 15-19)
   (c) No other conditional can be entered from such entry state.
4. Assume that the loop iteration starts in state (ulimit = lbuf + 13 $\wedge$ quotation = T $\wedge$ rquote = T).
   (a) The loop enters state (ulimit = lbuf + 14 $\wedge$ quotation = F $\wedge$ rquote = F) if it executes the second conditional (lines 25-29)
   (b) No other conditional can be entered from such entry state.

Thus, all possible states of the loop are correctly captured by the invariant. In other words, $\forall n : P(n) \Rightarrow P(n+1)$

□

On the other hand, the formula is violated when line 28 is absent from the loop. This shows that, when correctly queried, the solver is able to differentiate between a correct program and an invalid program even when subtle conditions are modelled. However, the loop invariant has to be provided manually. To our knowledge, there is no tool available to the public that would be able to infer such conditions automatically. Abstract interpretation techniques [12] based on control-flow partitioning [25] have shown to be useful for synthesizing loop invariants but synthesis of such complex invariants seems out of reach due to the over-approximation employed by abstract interpretation to keep full automation. Another strategy to infer simple loop invariants is to generate the candidates using invariant synthesis strategies based on simple grammars. Such approach has been used to perform runtime invariant synthesis and discover likely invariants [27] in software programs based on witnessed executions. We believe that this latter approach would not bring the desired result on this example since the invariant is not respected in the presence of a security vulnerability and thus would not be discovered by executing the program trace on which the vulnerability is to be found.

In order to illustrate the inference problem better, let us try to verify a more abstract formula that is a relaxation of the real invariant. Such simpler formula is an interesting candidate invariant for the loop as it contains less sub-formulae and thus is more likely to be generated automatically.

$$(ulim = lbuf + 15 \wedge quotation = F \wedge rquote = F) \\ \vee (ulim = lbuf + 14 \wedge (quotation = T \vee rquote = T)) \\ \vee (ulim = lbuf + 13 \wedge quotation = T \wedge rquote = T)$$

This second formula corresponds to the automaton in Figure 3. We do not indicate the input vocabulary of this second automaton as it does not correspond to a concrete representation of the loop, but a candidate abstraction of it. Unfortunately, this second invariant is not provable due to the introduced spurious state (T,T,14) which is not a real behavior of the loop. When starting the loop in such spurious state and executing the second conditional code of the loop (from line 20 to 24 on Figure 1), another spurious state (F,T,15) can be reached. Such state is violating invariant 2. Thus, invariant 2 does not hold at every iteration of the loop. This failed example shows the difficulty of abstracting information from an invariant without losing soundness.

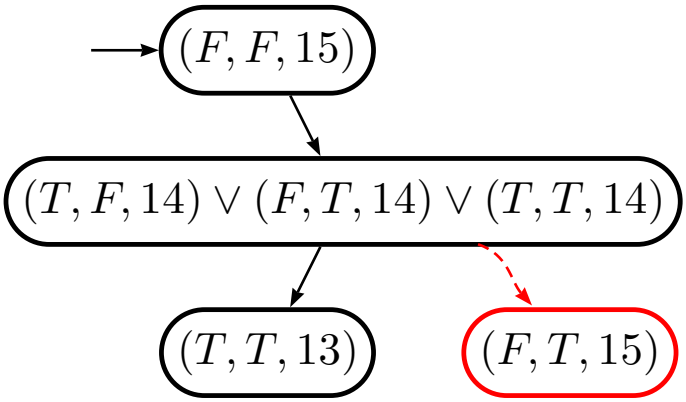

Figure 3: Simpler automaton that fails to model loop

The case study of this section illustrates well the limits of SMT solvers in absence of a proper constraint generation engine to feed them. It is possible to verify code invariants as long as those invariants are provided by the developer. Yet, automated analysis of such loop constructs is extremely challenging when no input from the developer is available, since generating the expected contract from a piece of code is not the role of the solver, and existing inference techniques are usually unable to cope with such complex loop invariants. Fortunately, there also exists many other properties [2] [23] [38] for which the contracts can be more easily guessed. We will see in the next section that such limitations are not specific to the scenario of vulnerability discovery.

3 SMT in Exploit Generation

Since 2008 there have been a number of papers [6, 20, 1] in which attempts have been made to develop systems for *Automatic Exploit Generation* (AEG) that rely on an SMT solver for constraint solving. This early work generally took the definition of an *exploit* to be an input to a program that, through leveraging memory corruption of some kind, results in the hijacking of the program counter and the execution of attacker controlled code. At their core, these systems are extensions of the input generation techniques that have successfully been applied to vulnerability detection [19, 18, 8, 7].

While they have had limited success in synthesizing exploits for simple vulnerabilities, with relaxed operating system security measures, there is a large theoretical and practical gap still to be bridged before they are applicable to real world problems. In this section we will explain how this gap results from primitive modeling of the

problem domain, rather than a limitation of SMT-based technologies. We will also discuss some of the more successful applications of SMT-based technologies to problems found in exploit development.

3.1 Restricted-Model Exploit Generation

Assume that we have a standard symbolic/concolic execution environment, such as S2E [10], BAP [5], TEMU [35], KLEE [7] or those described elsewhere [20, 19]. With any of these systems we can pause symbolic execution and for any memory address or register identifier retrieve the *path condition* for the data at that location. The path condition pc is a logical formula describing the constraints and manipulations performed on that data between its introduction from an attacker controlled source and the current point of execution. Consider the following sample x86 assembly code, with the assumption that byte in the AL register is under attacker control:

```
0:  add al, al
1:  sub al, 0x0f
2:  test al, al
3:  jz 6
4:  ...
5:  jmp 7
6:  ...
```

If we represent the input byte as b_0, and create a new variable b_n on each write to a variable, then at address 6 the path condition for the byte in AL is the following conjunction of clauses

$$b_1 = b_0 + b_0 \wedge b_2 = b_1 - 15 \wedge b_2 = 0$$

whereas at address 4 the path condition is

$$b_1 = b_0 + b_0 \wedge b_2 = b_1 - 15 \wedge b_2 \neq 0$$

One can then use a SMT solver to ask queries about the states represented by these formulae by appending constraints and looking for satisfying assignments. For example, if we wanted to check at address 4 whether the value 11 can be in the AL register we would create the formula:

$$b_1 = b_0 + b_0 \wedge b_2 = b_1 - 15 \wedge b_2 \neq 0 \wedge b_2 = 11$$

An SMT solver will then return a satisfying assignment, if one exists, such as $b_0 = 13$ in this case.

Effectively, the sum total of knowledge possessed by the system can be expressed as a map K from a collection of register/memory identifiers $(i_0, i_1, ..., i_n)$ to a path condition for each $(pc_0, pc_1, ..., pc_n)$. K has type $K : I \rightarrow F$. I is the union of the set of register identifiers, one for each register and subregister, with the set of identifiers for all valid memory addresses, one for each address. F is the set of closed quantifier free formulae over the theory of fixed sized bitvectors.

Exploit generation systems up to now have relied entirely on K in conjunction with a set of ad-hoc exploit templates to extend the work performed for input generation to produce exploits. We will refer to this approach as *restricted-model exploit generation*. The lack of success of such systems in tackling non-trivial exploits can be directly attributed to the restricted model of the execution environment used. Before discussing why this is, let us first define what we mean by an *exploit template* and then look at the two typical approaches to AEG.

3.1.1 Exploit Templates

A set of exploit templates T is a collection of algorithmic descriptions of basic strategies for taking advantage of vulnerabilities, that meet a set of criteria, in order to execute malicious code. A template $t \in T$ will take as input K and produce a SMT formula f. A satisfying assignment for f will be an exploit for P if the model encoded in f accurately models the constraints imposed on program inputs and any other relevant program state and environment properties. An AEG system will typically include several of these templates and select among them based on information derived from K and other information available about the type of vulnerability. As an example, if the AEG system detects, based on K, that on the execution of a `ret` instruction the memory pointed to by the ESP register is attacker controlled it would likely select a template that expresses the following constraint:

$$K(m) = v_0 \wedge K(m+1) = v_1 \wedge K(m+2) = v_2 \wedge K(m+3) = v_3$$

where m is the value in the ESP register and (v_0, v_1, v_2, v_3) are each values in the range 0-255 specifying the address we wish to redirect control flow to. Overall the strategies encoded in exploit templates so far have been rudimentary. Due to the fact that the only information source for accurate constraints on the program's state is K, the templates can only generate formulae that ask questions about the possible value ranges of bytes for which we have path conditions. Complex exploit strategies, such as those usually required to deal with modern binary hardening and OS protection mechanisms, require one to reason about a far richer domain than that described by K. For example, the state of the heap and its relationship with user input. Similarly, the exploitation of other vulnerability categories requires further abstractions and models to be introduced into the

symbolic/concolic execution phases of AEG. Use-after-free issues, for example, require both a description of the heap state and also higher level abstractions that entail objects and their allocation status.

3.1.2 AEG with Restricted Models

Two similar mechanisms have been used for AEG, both of which can be classed as *restricted-model exploit generation*. In the first [20], we start with a program P and an input I that we know to be *bad*. For example, I might have been produced by a fuzzer and causes P to crash. Under this approach we execute $P(I)$ within a concolic execution environment up until the point where a crash would occur during normal execution. At this point, K is available and we also have information on the cause of the crash e.g. we know if it was due to an attempt to execute, read or write invalid memory.

Such a system will also include a library of exploit templates T as described above. Using K and the knowledge of the cause of the crash, one or more templates can be selected and a set of formulae are generated.

The second approach integrates the vulnerability checking process with AEG [1]. Under this approach a symbolic/concolic execution environment is used to execute the program on symbolic input data. A safety predicate ϕ is invoked at particular program points to check whether a potentially unsafe operation is occurring. For example, on a `memcpy` call the safety predicate might check that the size argument is sufficiently restricted to prevent an overflow of the destination buffer. Once a safety predicate returns false this approach again has access to K but with the possibility of checking multiple possible exploit scenarios that depend on the value of elements of K.

For example, on a vulnerable `memcpy` assume that the destination buffer is n bytes in size but the size parameter can be m bytes, with $m > n$. Then there are $m - n$ possible lengths that would violate the safety property. Under the first approach a bad input provides a single violation of the safety property. In this case, each different input length $n < l \leq m$ results in the corruption of a different amount of data. Thus, for each value of l a set of formulae may be generated using K and T. Depending on the value of l the data corrupted could lead to different possibilities for exploitation, e.g. corrupting 4 bytes might lead to control of a pointer used as the destination of a write, while corrupting 12 might lead to control of a function pointer. The second approach can therefore generate more exploit candidates using a wider variety of templates. It is important to note though that each of these candidates is generated using K and T and is thus limited as previously described.

Conceptually, both of these approaches are quite close to the technology required for vulnerability detection systems based on symbolic/concolic execution. As such it is instructive to ask, "*Why does this approach succeed for vulnerability detection in real world scenarios but fail for exploit generation in real world scenarios?*".

The answer can be found by considering the accuracy of the model used in reasoning with respect to the problem being modeled. Successful vulnerability detection systems have found a wealth of vulnerabilities as a result of unsafe arithmetic within programs. The information required to accurately decide whether a sequence of instructions is unsafe or not, in this context, is contained within the path condition for the output bytes. In this case the model K is quite close to an ideal model for deciding questions of the problem domain.

If we ignore binary hardening, such as stack canaries, and OS security measures, such as address space layout randomization (ASLR), no-execute (NX) permissions on memory regions and more secure memory allocators, then AEG is also a tractable problem using the model K for certain vulnerability classes. In this environment the primary factor impacting the success or failure of an exploit is the manipulations and constraints imposed by the executed instructions. As these factors are modeled by K then it is possible for a template to create a SMT formula that accurately describes the requirements for a working exploit.

Some AEG systems account for certain protection mechanisms, such as limited ASLR without NX [20] and NX with limited ASLR [32], in conjunction with support for a limited set of vulnerability types, such as stack based overflows without functional stack hardening. The restrictions imposed by these systems on the problems they can handle effectively reduce the state of the exploited programs environment to one that is sufficiently accurately modeled by K.

The accuracy of K as a model rapidly deteriorates once one begins to consider correctly implemented protection mechanisms as found in modern Linux and Windows operating systems. It also deteriorates once one begins to consider exploitation of vulnerabilities that require the manipulation of environmental factors such as the heap layout. In these scenarios a useful model must account for the effects of user input on the memory layout.

3.1.3 The Future of AEG

In order to become a practical solution much work remains to be done in AEG. SMT-backed approaches have shown promise but the constraint generation phase still restricts the applicability of these systems to modern software and operating systems. In particular the model of the program and its environment must be improved.

Alongside this, it is worth considering the changes that will have to be made to the template driven approach to AEG in order to fit in with the trend towards more application and vulnerability-specific exploitation techniques.

Closing the gap between the model of a system and the system can be done in two ways. The first is the approach that has been taken so far with AEG, to reduce the complexity of the system until the model is sufficiently accurate. The second is to increase the sophistication of the model so that it entails more information on the system.

The reason the former approach was taken in the early work on AEG is straightforward; it is relatively cheap to extend previous work on symbolic/concolic execution to produce the model K. It is apparent that this model is an unrealistic abstraction of the state that must be taken into account for modern exploit generation. This can be seen if we consider the lack of any information on the relationship between the program's input and the layout of heap memory. Without such information we cannot generate constraints for which a satisfying assignment can manipulate the heap accurately. As a result, we cannot perform reliable AEG for any vulnerabilities that may be impacted by heap randomisation. This includes heap overflows but also use-after-free vulnerabilities which are the most prolific form of security flaw in web browsers.

When considering future directions for AEG it is important to look at the latest developments in manual exploit creation. For quite some time, the era of generic exploitation techniques that take advantage of obvious flaws in allocators and protection mechanisms has been drawing to a close. While there will always be exceptions, it is more common than not for exploits to leverage application and vulnerability-specific methods to avoid protection mechanisms than attempt to defeat them. For example, on a heap overflow it is far more likely to be successful if a controlled overwrite can be made to a pointer value within the same chunk that will later be called than attempting to corrupt heap metadata.

Modern exploits are also far more likely to leverage information leakage attacks, a topic that has so far received no attention in terms of AEG. While many may consider information leakage to result from different vulnerability types it is common for certain vulnerabilities, such as use-after-free, double-free etc, to be leveraged for both information leakage and code execution.

Both of these issues combined call into question the potential of template driven AEG to be sufficiently general to be useful. We consider it likely that once sufficiently accurate models are available it will be more useful to allow user-driven constraint generation based on their knowledge of exploitation, the application and the vulnerability in combination with more limited general and application specific templates.

3.2 Automated Payload Creation

While AEG systems have attempted to automate the control flow hijacking part of exploitation there has also been research on the application of SMT-based systems to the generation of ROP payloads [29, 34, 16, 32]. These systems have been developed to free an exploit developer from the tedious process of pouring over the potentially hundreds of thousands of candidate gadgets that may be found within a large binary.

As mentioned in the introduction, the approaches taken have fallen into two categories. Those that attempt to prove equivalence between a single gadget at a time and a model of some computation we wish to perform [34], and later systems that have attempted to provide a full compiler that can assemble multiple gadgets in sequence to achieve this computation [16, 32].

In the former approach, we first collect every valid sequence of instructions in the binary that ends in an instruction that can successfully transfer execution to the next gadget in sequence e.g. a `ret` instruction if the gadget addresses are provided at the location pointed to by ESP. The system takes as input these candidate gadgets and the specification of a computation s we require to be performed e.g. `ESP <- EAX + 8`, which specifies we are looking for a gadget that puts the value stored in the EAX register plus 8 in the ESP register.

First, the system will create an SMT representation s_{smt} of the computation specification. In other words, it will convert the specification to an SMT formula. The system will then perform a number of heuristic, but sound, reductions of the candidate gadget set e.g. eliminating any gadgets that neither read EAX or write ESP. At this point the system will iterate over the remaining candidate gadgets C and for each gadget $g \in C$ create the conjunction of a set of formulae that express the semantics g. For each $g \in C$ the formula $g \Leftrightarrow s_{smt}$ is then created and checked for validity. If the formula is *valid* it means that under all interpretations of the variables in g and s_{smt} their semantics are equivalent. This tells us that the gadget can be used to express the computation we require. If the formula is satisfiable but not valid it means that the gadget may work under some interpretations but may not under others. This would not be a desirable property of a component in a reliable exploit.

This approach can be quite useful in quickly discovering gadgets for simple computation specifications, such as the example given. However, such a system can only check if there is an exact correspondence between one gadget and the specification. Ideally, we would like to check whether the computation can be performed by chaining together n gadgets if necessary. This is often a requirement once our specification requires more complex data movement or arithmetic. A collection of m

gadgets can potentially be arranged in $m!$ different ways. Discovering the most useful potential combinations and then reasoning about their semantics requires a combination of heuristics and formula solving.

Systems have been developed and successfully applied to this problem, which have used SMT technology for different purposes. In [16], an SMT solver was used to reason about gadgets that contained branches. For example, if the gadget contained a branch an instruction sequence that might result in a crash a solver would be employed to check if it is possible that branch is never taken given the gadget's semantics. In [32], an SMT solver was used to look for arrangements of gadgets that meet the requirements of the computation specification.

Both the single-gadget and gadget compiler approaches have been successful at alleviating a certain amount of manual effort in the process of ROP payload creation. In both approaches we can see the pattern that exists throughout most successful integrations of SMT technology — minimizing the number of queries that must be made to a solver, reducing the problem space through the use of less computationally expensive algorithms and ensuring the constraints generated are a sufficiently accurate model of the problem being reasoned about.

4 SMT in protection analysis

Software protection analysis is critically important in dealing with malware, since most samples employ some sort of packing or obfuscation techniques in order to thwart analysis. It is also an area of economic concern in protecting digital assets from piracy and intellectual property theft. We present several areas in which SMT solvers have been practically applied towards these problems.

4.1 Equivalence checking for verification of deobfuscation results

Virtualization obfuscators [31] are an especially complex category of software protection tools that are commonly abused by malware. These tools work by converting portions of the program-to-be-protected's x86 machine language into a randomly-generated language that is then executed at runtime in an interpreter, which itself is also randomly-generated and obfuscated. The original x86 machine code in these regions is then overwritten. Code is generated within the binary such that, at run-time, when the program goes to execute the protected code, the register state is saved to some location (e.g., onto the stack), the interpreter executes, the register state is reloaded, and then the unprotected portion of the program executes normally. The end result is that the malware analyst must either possess a tool to invert the translation, or reverse engineer the code as it is running inside of the interpreter (rather than in the form to which he or she is more accustomed, viz., x86 code).

Such tools make life difficult for the reverse engineer, and they are also quite complex for the protection author to construct. The translation must capture precisely the semantics of the instructions under consideration, otherwise the virtualized program has the potential to produce different behaviors than the original, which – by explicit design goal – would be very difficult for the developer to diagnose. Given this complexity, it is not a surprising notion that these tools might have bugs in the form of improper translations. Similarly, if the analyst were to construct an inversion tool to deobfuscate a virtualization obfuscator, the complexity of such a tool could easily lead to bugs in the form of improper deobfuscation.

Equivalence checking is a well-known technique for verifying the equivalence of two pieces of code. The simplest case is when the code snippets are straight-line (i.e. branchless). As an example, consider the C programs $x_0 = y + y$; and $x_1 = y << 1$;. Both of these programs logically encode the notion of doubling the variable y and storing the result in some other variable (since shifting left by one corresponds to multiplying by two). To determine whether these sequences produce equivalent results, we encode them as SMT formulae and then query the decision procedure for the condition $x_0 \neq x_1$. If this formula is satisfiable, the SMT solver will return a counterexample, namely, a value of y for which the sequences differ. If this formula is unsatisfiable, this is a proof (assuming the soundness of the solver) that the two sequences always produce the same output, given the same input.

To apply this procedure to two branchless sequences of x86 instructions, we convert both sequences to our intermediate representation, then put both sequences in Single Static Assignment (SSA [13]) form, convert the SSA version of the IR to SMT formulae, and query the decision procedure as to whether the output variables (i.e., flags, registers, and memory) can ever differ (i.e., $eax_{seq1} \neq eax_{seq2} \vee ebx_{seq1} \neq ebx_{seq2} \ldots$). To compare the contents of memory in this way, the solver must support the theory of extensional arrays (which many modern solvers fortunately do).

SMT-backed equivalence checking provides a powerful primitive for ensuring the correctness of a deobfuscation procedure on branchless sequences. One simply generates some branchless program that falls within the purview of the virtualization obfuscator, obfuscates it, deobfuscates it, and uses equivalence checking to compare the resulting code against the original code.

Applying this procedure helped discover potentially

incorrect translations in TheMida CISC VM [37] after having constructed a deobfuscation procedure for this protection. The virtualization of certain instructions such as *ror* and *inc* did not take some of the subtleties of those instructions into account. In the case of *ror* and similar instructions, the Intel manuals dictate that these instructions do not modify the flags if the shiftand is zero. Therefore, improper maintenance of the flags prior to the execution of these instructions could cause the flags to take different values in the obfuscated and deobfuscated versions. Similarly, "inc" does not modify the carry flag, so any modification to this flag induced by the obfuscator before the instruction executes would result in incorrect machine state.

Figure 4 illustrates a more subtle example of potential incorrectness in translation. This (deobfuscated) instruction sequence loads an address (stored in enciphered form) from the memory location pointed at by the esi register, deciphers the address in the next four instructions, then loads a byte from that address and pushes it onto the stack. Since the ciphering process is invertible, this code snippet enforces no restriction upon the range of addresses from which the byte could potentially be loaded. Therefore, the address could well point onto the stack, below the location of the current stack pointer. Since the obfuscator introduces many spurious writes to the stack, the value loaded in the deobufscated world could differ from the one in the obfuscated world. This translation error would be unlikely to result in a runtime error in the real world, but it demonstrates the exhaustive capabilities of SMT solvers towards the equivalence checking problem.

```
1:  lodsd dword ptr ds:[esi]
2:  sub eax, ebx
3:  xor eax, 7134B21Ah
4:  add eax, 2564E385h
5:  xor ebx, eax
6:  movzx ax, byte ptr ds:[eax]
7:  push ax
```

Figure 4: Deobfuscated sequence

4.2 SMT-based input crafting for semi-automated cryptanalysis

When it comes to constructing licensing systems, best practices dictate that only properly-vetted implementations of trusted cryptographic algorithms be utilized as part of securely-designed cryptosystems. (Even then, that this may be insufficient to prevent against certain types of attacks such as those where the attacker is able to replace private keys within a binary or otherwise patch the program's logic. As part of the never-ending cat and mouse game in software protection, techniques such as [39] can be used to obfuscate cryptographic keys, and techniques such as [36] demonstrate that these techniques are not necessarily infallible). Many protection authors unfortunately still do not heed this advice, leading to a glut of cracked software available on peer-to-peer networks.

SMT solvers can be utilized as a medium for manually modelling licensing schemes. [14] focuses on one scheme in particular, which is partially depicted in Figure 5.

```
1:  again:
2:   lodsb
3:   sub   al, bl
4:   xor   al, dl
5:   stosb
6:   rol   edx, 1
7:   rol   ebx, 1
8:   loop  again
```

Figure 5: The main loop of the serial algorithm

First, the authors manually cryptanalyze the protection from an algebraic standpoint and construct a highly efficient key generator. Next, the authors demonstrate how to manually model the scheme in terms of an instance of the SAT problem. Since modern SMT solvers subsume SAT solvers, the scheme can obviously be manually modelled in terms of operations within the bitvector theory, in a manner that is more succinct and natural than the low-level bitwise manual CNF encoding. For instance, a multiplication operator can be modelled natively as one term within many solvers, whereas to model such a thing in terms of operations upon individual bits (as in a SAT instance) generates complex circuits. One iteration of the loop (particulary, iteration i) as shown in Figure 5 can be manually modelled in SMT as follows:

$$
\begin{aligned}
& al_{0,i} = activation\_code[i] \\
& \wedge\, al_{1,i} = al_{0,i} - ebx_i[7:0] \\
& \wedge\, al_{2,i} = al_{1,i} \oplus edx_i[7:0] \\
& \wedge\, output[i] = al_{2,i} \\
& \wedge\, edx_{i+1} = rotate\_left(edx_i, 1) \\
& \wedge\, ebx_{i+1} = rotate\_left(ebx_i, 1)
\end{aligned}
$$

In this formula, $activation\_code$ corresponds to the memory region pointed at by the esi register and is an input to the serial algorithm, $output$ corresponds to the memory region pointed at by the edi register, and $rotate\_left$ is a built-in function in many SMT solvers for performing leftward rotation.

The same technology that is used in other problem domains for more conventional tasks in programming language theory, such as those discussed hereinbefore i.e.

vulnerability discovery and test-case/exploit generation, can also be repurposed for the sake of solving problems such as this one semi-automatically. The Pandemic binary program analysis framework was employed [30] to automatically (statically) generate an execution trace of a run of the algorithm, where the user's input is treated as free variables. We then manually constructed the post-condition that the output must satisfy, and then fed the results to an SMT solver. The inputs derived by the solver correctly break the scheme. Space considerations force us to refer the interested reader to [30] for more details of the problem, the system architecture of Pandemic, and the solutions.

These problems can be attacked purely statically, if the analyst is willing to invest the time in manually modelling the state required to simulate the execution of the serial algorithm, or in a concolic fashion (which allows for greater automation). Static solutions may be preferred when the analyst wishes to investigate the properties of some piece of code that he or she might not know how to trigger; a static investigation may inform the analyst whether such an undertaking is merited (i.e., whether the portion of code exhibits some vulnerability; if this is not the case, then the broader problem of driving execution to that location would be fruitless).

This particular problem instance has the nice property that the path that the algorithm takes is not dependent upon the user's input. Specifically, the algorithm consists of a loop that executes for a fixed number of iterations before comparing the output to a fixed value. Hence, the problem is easier to solve than what might be the case if the path were input-dependent, for example, if multiple checks lead to failure cases, or if the input length were unbounded and the algorithm iterated over it in its entirety (such conditions could potentially be mitigated through the use of loop invariants, perhaps automatically-synthesized ones). We emphasize, however, that there is nothing special about serial algorithms that place them in a strictly restricted class of the general input-crafting problem: any type of code (including obfuscated code) with unrestricted programming language constructs might be utilized to implement a serial check – and in fact, the constraints might even be harder than the ordinary case due to the prevalence of hard cryptographic operations. Hence, progress towards this pursuit is tied to progress in binary program analysis and verification/SMT solvers in general. Nevertheless, these early results are encouraging.

5 Conclusion

SMT solvers are becoming an integral part of the security engineer's tool kit. We presented three applications of SMT solvers in vulnerability discovery by static analysis, exploit generation (a specialization of input crafting), and copy protection analysis. In these three applications, solvers do a remarkable job of assisting the analysts in deciding whether suggested solutions are valid in their respective problem space. Yet, solvers are not suited for generating domain-specific problem descriptions as the preliminary constraint generation step has to be performed outside the solver. We expect that specialized constraint inference assistants will improve in the future and help generate formal problem definitions for non-trivial problems in the area of computer security.

6 Acknowledgements

The authors would like to thank Shuvendu Lahiri and Matt Miller for their insights on this article.

References

[1] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium* (Feb. 2011), pp. 283–300.

[2] BALL, T., HACKETT, B., LAHIRI, S. K., QADEER, S., AND VANEGUE, J. Towards scalable modular checking of user-defined properties. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments* (Berlin, Heidelberg, 2010), VSTTE'10, Springer-Verlag, pp. 1–24.

[3] BALL, T., LAHIRI, S., AND MUSUVATHI, M. Zap: Automated theorem proving for software analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning*, G. Sutcliffe and A. Voronkov, Eds., vol. 3835 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 2–22.

[4] BALL, T., LEVIN, V., AND RAJAMANI, S. K. A decade of software model checking with SLAM. *Commun. ACM 54*, 7 (July 2011), 68–76.

[5] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification* (Berlin, Heidelberg, 2011), CAV'11, Springer-Verlag, pp. 463–469.

[6] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 143–157.

[7] CADAR, C., DUNBAR, D., AND ENGLER, D. 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* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[8] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur. 12*, 2 (2008).

[9] CHESS, B. V. Improving computer security using extended static checking, 2002.

[10] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst. 30*, 1 (2012), 2.

[11] CLARKE, E. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar, Eds., vol. 1346 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997, pp. 54–56. 10.1007/BFb0058022.

[12] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1977), POPL '77, ACM, pp. 238–252.

[13] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (1991), 451–490.

[14] DCODER, AND ANDREWL. kaos "toy project" and algebraic cryptanalysis.

[15] DULLIEN, T. The future of exploitation revisited. *Infiltrate conference* (2011).

[16] DULLIEN, T., KORNAU, T., AND WEINMANN, R.-P. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX conference on Offensive technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–.

[17] FLANAGAN, C., AND LEINO, K. R. M. Houdini, an annotation assistant for ESC/Java. *FME 2001: Formal Methods for Increasing Software Productivity* (2001).

[18] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 213–223.

[19] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. SAGE: Whitebox fuzzing for security testing. *Queue 10*, 1 (Jan. 2012), 20:20–20:27.

[20] HEELAN, S. Automatic generation of control flow hijacking exploits for software vulnerabilities. Msc. dissertation, University of Oxford, September 2009.

[21] LAHIRI, S.K, Q. S. B. S. HAVOC: Heap aware verifier for c and c++ programs. `http://research.microsoft.com/en-us/projects/havoc/`.

[22] LAHIRI, S. K. Unbounded system verification using decision procedures and predicate abstraction. Tech. rep., Phd thesis, Carnegie Melon University, 2004.

[23] LAHIRI, S. K., AND VANEGUE, J. ExplainHoudini: Making Houdini inference transparent. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation* (Berlin, Heidelberg, 2011), VMCAI'11, Springer-Verlag, pp. 309–323.

[24] LEINO, K. R. M. The Boogie 2 project. `http://boogie.codeplex.com/`.

[25] MAUBORGNE, L., AND RIVAL, X. Trace partitioning in abstract interpretation based static analyzers. In *Programming Languages and Systems*, M. Sagiv, Ed., vol. 3444 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 139–139.

[26] NIKOLAI BJORNER, L. D. M. The Z3 constraint solver. `http://research.microsoft.com/projects/z3/`.

[27] PERKINS, J. H., AND ERNST, M. D. Efficient incremental algorithms for dynamic detection of likely invariants. *SIGSOFT Softw. Eng. Notes 29*, 6 (Oct. 2004), 23–32.

[28] POTTIER, F., AND REMY, D. The essence of ml type inference. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, 2005, ch. 10, pp. 389–489.

[29] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security 15*, 1 (Mar. 2012).

[30] ROLLES, R. Semi-automated input crafting by symbolic execution, with an application to automatic key generator generation.

[31] ROLLES, R. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX conference on Offensive technologies* (Berkeley, CA, USA, 2009), WOOT'09, USENIX Association, pp. 1–1.

[32] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium* (Aug. 2011).

[33] SENDMAIL, P. Sendmail release notes for the crackaddr vulnerability.

[34] SOLE, P. DEPLIB 2.0. Ekoparty 2010.

[35] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND 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, Dec. 2008).

[36] SYSK. Practical cracking of white-box implementations. `http://www.phrack.com/issues.html?issue=68&id=8`.

[37] THEMIDA, P. Themida. `http://www.oreans.com/themida.php`.

[38] VANEGUE, J. Zero-sized heap allocations vulnerability analysis. In *Proceedings of the 4th USENIX conference on Offensive technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–8.

[39] WYSEUR, B. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009.