Program Protection Through Software-Based Hardware Abstraction

J. Todd McDonald[1] [a], Ramya K. Manikyam[1] [b],
Sébastien Bardin[2], Richard Bonichon[3], Todd R. Andel[1]

[1]*Department of Computer Science, University of South Alabama, Mobile, AL, USA*
[2] *Université Paris-Saclay, CEA, LIST, France*
[3]*Nomadic Labs, France*

{*jtmcdonald,tandel*}*@southalabama.edu,rk1421@jagmail.southalabama.edu, sebastien.bardin@cea.fr,*
*richard.bonichon@nomadic-labs.com*

Keywords: Software protection, MATE attacks, virtualization, symbolic analysis.

Abstract: Software companies typically embed one or more secrets in their programs to protect their intellectual property (IP) investment. These secrets are most often processed in code through evaluation of point functions, where only the correct password, PIN, or registration/activation code will authorize an end-user to legally install or use a product. Man-at-the-End (MATE) attacks can break assumptions of program security to find embedded secrets because they involve legitimate software owners who have complete access to the software and its execution environment. In this research, we present a novel approach to software MATE protection that leverages gate-level hardware representation, namely *software-based hardware abstraction* (SBHA). As a new proposed form of virtualization for software protection, SBHA demonstrates a light overhead – especially compared to much costlier traditional virtualization transformations, while completely defeating almost all symbolic execution-based attackers that were studied. Overall, SBHA bridges the gap between hardware and software protection, paving the way for future developments.

1 INTRODUCTION

The software industry is one of the most important sectors of the global economy and has progressed tremendously in the past few decades. Intellectual property (IP) rights are an integral part of the software industry and have to be properly guarded; losses due to intellectual property theft can indeed negatively impact the global economy and even national security. Legitimate software companies are faced with a myriad of attacks such as software piracy[1], malicious reverse engineering and tampering attacks [Falcarin et al., 2011]. These attacks are generalized as Man-at-the-End (MATE) attacks where the attacker can be a legitimate end-user and has complete access to the execution environment [Collberg and Nagra, 2009]. A recent BSA study[2] placed the global piracy rate at 39% and financial losses due to software piracy and unlicensed software around $52.2 billion, with a sig-

nificant impact on the global economy. Hence the need for software IP protection.

Figure 1: Notional Interpreters for Software Protection
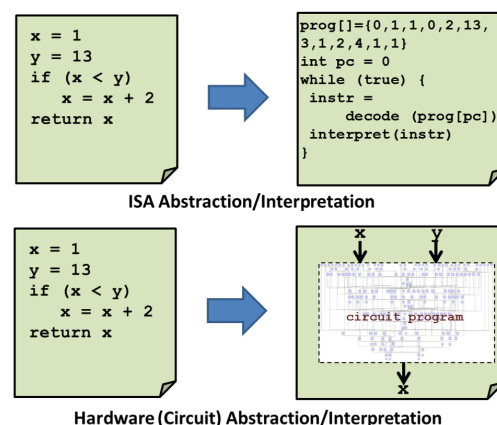
Software obfuscation. For companies that cannot rely fully on IP laws or that prefer a more proactive approach in combating MATE attacks, software obfuscation offers a technical solution. Apart from theoretic impossibility limits proving that no general

approach can achieve information theoretic security [Barak et al., 2012], practical transformation algorithms that drive up the time and cost required to illegally reverse engineer or crack legitimate software provide the most readily available defense [Schrittwieser et al., 2016]. Still, recent works [Banescu et al., 2016, Bardin et al., 2017] demonstrate that so-called semantic attacks, based on advanced program analysis such as dynamic symbolic execution (DSE) [Cadar and Sen, 2013], are highly effective against conventional obfuscation methods.

SBHA protections. This paper introduces and evaluates the effectiveness of a novel approach to software protection, called *software-based hardware abstraction* (SBHA), that leverages *virtualization of software constructs under the form of hardware-like constructs* to impede reverse engineering or hinder adversarial analysis. Virtualization essentially serves the role of an interpreter, converting one target program language into another. Virtual interpreters are often used in program protection schemes [Cheng et al., 2019], converting an original program into a new instruction set architecture (ISA), and then having the transformed program act as an interpreter – in our case a program that behaves like a CPU, but which has its own instruction set, code and execution stack. Fig. 1 illustrates a typical ISA abstraction transformation from code. We leverage the fundamental concept first pointed out by [Vahid, 2007] that "every program can be converted to a circuit and every circuit can be represented as a program" to implement a software protection technique called *Software-based Hardware Abstraction* (SBHA). We represent embedded program constructs as combinational circuit abstractions, but use software to represent that circuit logic and execute it (Fig. 1).

Contributions. Our contributions include:

- We propose SBHA as a novel, low-overhead obfuscation technique for transforming point functions in C programs to hardware abstractions (Sec. 4). SBHA provides a natural framework where standard software obfuscation can be combined with protections inspired from hardware circuit obfuscation (e.g., anti-SAT and anti-BDD), making deobfuscation even more challenging as it requires a dual software-hardware expertise;

- We demonstrate the dramatic *effectiveness* of SBHA against state-of-the-art attacks by DSE engines such as KLEE and Angr (Sec. 5), which had only 1 success out of 320 test samples, despite a 120 hour time-out per sample;

- We evaluate the *resilience* (i.e. robustness to adversarial reverse engineering) of SBHA in a com-

prehensive manner against worst-case scenarios in regards to circuit recovery attacks, compiler optimizations (representing deobfuscating attackers) and path merging attacks. Adding anti-BDD or anti-SAT methods from the hardware side dramatically boost resilience here;

- Finally we establish SBHA as a low *cost* technique with minimal overhead (Sec. 5) and medium *stealth* (given supporting obfuscations) transformation; our results establish SBHA as a new form of software virtualization that demonstrates only 10% runtime overhead, 3.5x source code and 50% executable code overhead (for point functions) – compared to much costlier traditional virtualization.

SBHA unifies two historically disparate research areas: software obfuscation and hardware (circuit) obfuscation. It provides a bridge to leverage more than a decade of prior work and integrate the best of both research communities in the context of improving software protection, paving the way to completely new software and circuit protection mechanisms. We provide additional benchmark data and descriptions at https://soc.southalabama.edu/~mcdonald/SBHA.

2 Motivation

We detail here our attacker model (Sec. 2.1) and present a motivating example (Sec. 2.2).

2.1 Attacker Model

We consider the general context of Man-at-the-end (MATE) attacks (also termed white-box attacks): the attacker has full access to the code under attack, but only under its binary form and not as source code, and seeks to discover program secrets embedded in binary executable programs. These secrets can take for example the form of embedded passwords, keys, PIN codes, etc.

Capabilities. We consider a skilled and economically motivated adversary [Ceccato et al., 2017], able to take advantage of state-of-the-art static and dynamic analysis tools such as decompilers, disassemblers, debuggers, tracers, slicers and emulators, among others. Yet, this attacker has a limited budget for the attack, and does not have the resource or will to develop "beyond state-of-the-art" tools. Hence, an effective strategy for the defender is to design protections making manual reverse complicated and at the same time breaking the best automated known attacks.

Regarding automated attacks, we focus on attackers with the capacity to use dynamic symbolic execution (DSE) engines [Cadar and Sen, 2013] to leak program secrets – the technique has been shown highly effective for deobfuscation [Schrittwieser et al., 2016, Banescu et al., 2016, Salwan et al., 2018]. Note that, in practice, DSE tools are correct (every discovered path is actually feasible) but *incomplete*, for they can be tricked into missing feasible paths [Yadegari et al., 2015]. Interestingly, our experimental evaluations (Sec. 5) consider some source-level tools as well, while program analysers on source code are more precise than on binary code [Balakrishnan and Reps, 2010]. Hence, *our experiments favor the attacker even more*, and, consequently, strengthen our protection results.

Program 1 Function with PWD/PIN Checks

```
void SECRET(unsigned long input[1],
            unsigned long output[1]) {;
  char pwd[100]  = "";
  int failed  = 0;
  int strCompareResult, pincode ;
  pincode = input[0UL];
  printf("Please enter password:");
  scanf("%s", pwd);
  strCompareResult =
    strncmp(pwd, "key$", 100UL);   //pwd
  failed |= strCompareResult != 0UL;
  failed |= pincode != 1123UL;     //pin
  if (failed) { output[0] = 0UL; }
  else {        output[0] = 1UL; }
}
```

2.2 Motivating Example

A typical attack scenario consists in recovering passwords and specific triggering input, either for themselves or for enabling further progress in the deobfuscation process. Let us give an example. Program 1 elaborates the call to a SECRET function containing a *point-function*[3] if-statement check for a password and activation code entered by the user. Results by [Banescu et al., 2016] and [Holder et al., 2017] showed that state-of-the-art (SOTA) tools such as KLEE[4] and Angr[5] can easily recover the expected secrets of such programs under analysis, even with a wide assortment of obfuscations from Tigress[6] and Obfuscator-LLVM [Junod et al., 2015] applied to the original program. These studies also showed that the addition of virtualization would drive up the run-time

---

[3]Function returning 0 or false for all input but one.
[4]https://klee.github.io/
[5]https://angr.io/
[6]http://tigress.cs.arizona.edu/

cost of KLEE and Angr, but no standard transformation combined with virtualization would absolutely defeat the analyses – while yielding a significant overhead to the protected program. For example, [Ollivier et al., 2019a] report virtualization runtime overhead as 1.5x increase for 1 level, between 15x and 50x for 2 levels, and between 100x and 1000x for 3 levels of virtualization. We show in this paper that SBHA completely defeats symbolic execution from KLEE and Angr (no recovery of the password within 120h) in such test programs, for only a negligible runtime overhead and reasonable 10% code size overhead.

3 Background

We provide a brief overview of obfuscation, semantic attacks and circuit abstractions.

3.1 Program Obfuscation

Obfuscation is a process of altering programs (software or hardware) in such a way that MATE attackers are hindered or prevented from analyzing, altering, or pirating the program [Collberg and Nagra, 2009, Collberg and Thomborson, 2002]. We can define an obfuscator $O : \mathcal{P} \to \mathcal{P}$ as a transformation taking as input a program P and producing a semantically equivalent version P', such that $P(x) = P'(x)$ for all input x, and P' is (expected to) be harder to analyze than P.

Given enough time, an obfuscated program can be reverse-engineered [Manikyam et al., 2016, Barak et al., 2012, Collberg and Nagra, 2009]. In general, programs are protected because of some inherent secret information contained in them: we define such information as a *property* of a given program (also known as a *program asset* [Basile et al., 2019]).

Evaluated qualities. In their seminal early work on defining software obfuscation, [Collberg et al., 1997] identify four key metrics:
M1 Effectiveness: Analysis and modification of P' should require more time than for the original program;
M2 Resilience: The ability to resist adversarial reverse engineering. Especially, the protection itself should be hard to defeat through automated means;
M3 Stealth: P' should have the same statistical properties as the original program;
M4 Cost: The execution time and overhead due to obfuscation should be minimized.

Especially, effectiveness captures the notion that good obfuscating transformations should increase resources (e.g., time) required for an analysis technique

or tool to reveal the secret property. In our case, we focus on the ability of a protection to protect a program against automated semantic deobfuscation techniques, namely symbolic execution.

3.2 Semantic Deobfuscation Techniques

So-called semantic or symbolic deobfuscation techniques rely on advanced (semantic) program analysis methods such as abstract interpretation [Rival and Yi, 2020] or symbolic execution [Cadar and Sen, 2013] to overcome or simplify obfuscated constructs within a protected program. Several existing work have established the effectiveness of such methods against standard protection schemes [Banescu et al., 2016, Salwan et al., 2018, Bardin et al., 2017], especially in an interactive attack scenario where the attacker launches local automated attacks on well-chosen parts of code. See [Schrittwieser et al., 2016] for a survey . We focus especially on Dynamic Symbolic Execution [Cadar and Sen, 2013], whose ability to combine both the robustness of dynamic analysis (useful for, e.g., bypass packing or self-modifying code) and the reasoning-ability of symbolic methods (useful for, e.g., find triggering inputs and cover the path space) makes it a weapon of choice.

3.3 Hardware and Circuits

Combinational circuits implement Boolean logic functions directly through a set of potential logic gates (referred to as the basis set Ω) such as *AND*, *OR*, *XOR*, *NOT*, *NAND*, *NOR*, and *XNOR*. Structurally, circuits can be expressed in a number of ways including textually in netlist languages such as BENCH format [Hansen et al., 1999] and visually in schematic form. Fig. 2 illustrates a small 5 input, 2 output, 6 gate combinational circuit in schematic form with corresponding BENCH netlist. A semantic truth table represents the behavior of a *n*-input, *m*-output circuit where a vector of input bits derives a vector of output bits, corresponding to a set of Boolean functions $f_i : B^n \rightarrow \{0, 1\}$, where $i = 1..m$ [De Micheli, 1994]. Each row of a truth represents one of the 2^n input vectors and its corresponding output vector, with a subset of I/O vectors illustrated in Fig. 2.

4 Software-Based Hardware Abstraction

To introduce software-based hardware abstraction (SBHA) and evaluate its potential as a software protection technique, we first examine its usefulness in
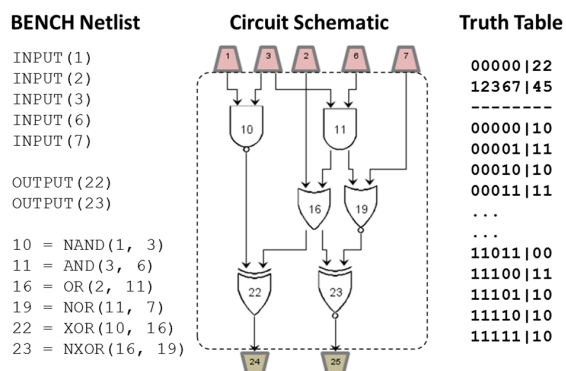

Figure 2: Equivalent Circuit Representations

protecting point functions which are commonly used to protect program access to only authorized, legitimate end-users. Point functions typically are conditional constructs in software. In this section, we provide a simple walk-through to illustrate the transformation process using C (source) to C (source). Basic SBHA transformation requires three steps:

1. Given a program *P* in source code form, first identify the relevant point function software construct.

2. Virtualize the software construct into a semantically equivalent Boolean logic form, then synthesize it into a combinational logic circuit (*C*).

3. Implement the hardware construct (*C*) into source code form and replace it in the original program, creating an SBHA variant P_{HW}.

Program 2 1-Char Password C Program

```c
int main( int argc, char *argv[] )  {
  int compareResult = strncmp(argv[1], "%", 1);
  if (compareResult != 0) {
    printf ("0"); exit(-1);
  }
  else {
    printf ("1"); runprogram();
  }
}
```

Companion material with program code listings and the full set of benchmarks is located at https://soc.southalabama.edu/~mcdonald/SBHA.

4.1 Basic Algorithm

As an example, we illustrate a C program that takes as input a 1-character password, seen in Program 2. The program checks to see if the password matches and outputs 0 then exits if it does not, and otherwise outputs 1 and performs the remainder of the program functionality otherwise. This example is abstract in

the sense that normally passwords are verified against cryptographic hashes, but nonetheless, only the correct password will allow the program to run correctly.

Given a software construct, such as an if-then-else point function, a corresponding truth table or Boolean function can be used to represent the semantics of the construct. Fig. 3 illustrates 1. the corresponding truth table; and 2. the corresponding circuit structure/netlist that implements the function embodied by the conditional statement of Program 2. From a Boolean function or truth table, a circuit structure can be derived using Sum-of-Products (SOP or disjunctive normal form) or Product-of-Sums (POS or conjunctive normal form) synthesis. In the example program, only one input (the ASCII character %) produces true ASCII characters are 8-bit values:% is 0x25. or 0b00100101. In essence, the software construct for the conditional statement can be represented as a 8-input/1-output combinational circuit. Fig. 3 shows that the SOP form of the truth table is directly translated into a circuit. For each minterm or product term (an input that produces a 1 output), the circuit has an AND gate (a Boolean product) with inverted inputs (NOT gates) for each 0 bit and normal inputs for each 1 bit in the input vector. Since point functions have only one minterm, there is a single AND gate in the circuit. In SOP form, 0b00100101 translates to !8 * !7 * 6 * !5 * !4 * 3 * !2 * 1, where numbers correspond to numbers of input signals, and ! to inverted input values.

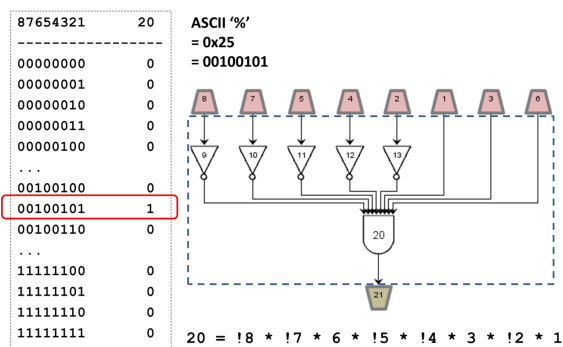| 87654321	20	ASCII '%'
----------------		= 0x25
00000000	0	= 00100101
00000001	0	
00000010	0	
00000011	0	
00000100	0	
...		
00100100	0	
00100101	1	
00100110	0	
...		
11111100	0	
11111101	0	
11111110	0	
11111111	0	20 = !8 * !7 * 6 * !5 * !4 * 3 * !2 * 1

Figure 3: Circuit for Checking Character %

Given the circuit (C) that represents the function of the software construct embodied in program P, SBHA then translates this back into software form to produce the variant P_{HW}. To accomplish this, we created and used a custom Boolean logic library (logiclib) that has corresponding C code for each logic gate type. For gates with fan-in greater than 2, there are functions allowing the evaluation of multiple (more than 2) inputs. In the C program, Boolean values correspond directly to the logic values computed by functions of Boolean logic gates found in a circuit structure.

Program 3 SBHA Point Function Program

```
void runcircuit(bool input[], bool out[]) {
  bool v8 = input[0], v7 = input[1],
  ... v2 = input[6], v1 = input[7];
  bool v9 = not(v8), v10 = not(v7),
  v11 = not(v5), v12 = not(v4), v13 = not(v2);
  bool v20in[8];
  v20in[0] = v9; v20in[1] = v10;
  v20in[2] = v6; v20in[3] = v11;
  v20in[4] = v12; v20in[5] = v3;
  v20in[6] = v13; v20in[7] = v1;
  v20  = andmulti(v20in,8);
  out[0] = v20;
}
```

The translation of the circuit results in C code that is now semantically equivalent to the conditional statement in the original program. Program 3 shows the equivalent C code, embodied in a function runcircuit. Input and output takes the form of bool arrays, with the encoding of the circuit directly represented by logiclib functions.

Program 4 Final SBHA Variant Code

```
int main( int argc, char *argv[] )  {
  bool circuitinput[8];
  bool circuitoutput[1];
  convertString(argv[1],circuitinput,1);
  runcircuit (circuitinput, circuitoutput);
  if (circuitoutput[0]) {
    printf ("0"); exit(-1); }
  else {
    printf ("1"); runprogram(); }
}
```

In order to utilize the runcircuit function, the original character input of the C program must be converted into a Boolean value array representation. We accomplish this translation using a conversion function that takes as input the original character string variable (here argv[1]) and the input size in characters (in our example, 1 character), and initializes a provided Boolean array with the corresponding ASCII bit values of the corresponding character string. Program 4 shows the corresponding C code for input conversion in our example program. The SBHA transformation is thus a source-to-source translation that virtualizes software constructs in program P and produces a variant P_{HW} that is semantically equivalent. As Program 4 illustrates, the output of the SBHA virtualization can be directly used to provide supporting functionality consistent with the original code.

4.2 Extended Algorithm

The basic SBHA algorithm involves the straight translation of software constructs into hardware equivalent forms, while still retaining software source code representation. In this paper, we answer partially the ramifications of hardware-based virtualization on traditional program analysis tools and techniques where SBHA performs a straight translation of point functions in the standard three step process $P \rightarrow C \rightarrow P_{HW}$. Fig. 4 illustrates how SBHA will bring unification of two disparate fields of research: circuit/hardware (HW) obfuscation and software (SW) obfuscation. We analyze first the effects of standard SBHA, but also consider the possibility that 1) the hardware netlist representation (C) can be further transformed via gate-level circuit obfuscation algorithms (to produce variant C') and 2) the SBHA source variant (P_{HW}) can be further transformed via software obfuscation algorithms to produce a final variant P'. Hardware specific protections include:

1. Anti-BDD: one of the basic adversarial techniques is to use Binary Decision Diagram (BDD) reduction to express logic components into compact functional representations. However, BDD complexity and construction overhead [Woelfel, 2005, Beyer and Stahlbauer, 2014] depends on the structure of the underlying circuit, and thus avails itself to countermeasures. Our anti-BDD technique uses multiplier circuit constructs known to have an exponential lower bound for BDD encoding [Bryant, 1991, Woelfel, 2005];

2. Anti-SAT: a well-known technique in hardware deobfuscation is the use of (Boolean) Satisfiability Solvers (SAT-solvers) for representing logic networks to find reduced functional expressions or solve unknown values. Our anti-SAT algorithm is based upon [Subramanyan et al., 2015]'s, which was suggested as a possible means to improve resistance to logic encryption attacks ((see technical report mentioned Page 4 for more details).
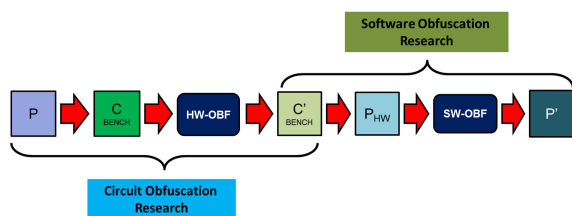

Figure 4: Extended SBHA Implementation

SBHA stands alone as a new form of obfuscating transformation, but has a higher potential for resilience and stealth when combined with both SW and HW obfuscations. We discuss our evaluation methodology and key metrics for SBHA next.

5 Experimental evaluation

We seek to experimentally assess the four following Research Questions:

RQ1 What is the effectiveness of SBHA transformations against DSE attacks?

RQ2 What is the resiliency of SBHA transformations against adversarial reverse engineering?

RQ3 What is the stealth of SBHA transformations when used in typical code?

RQ4 What is the cost (runtime and code size) overhead induced by SBHA transformations?

5.1 Computing Environment

We developed a custom virtualized container-based tool named Argon https://github.com/elm3nt/argon-cli which contains KLEE version 2.0 and Angr version 8.19.4.5. Experiments for **RQ1** were executed using a dense memory cluster (DMC) provided by the Alabama Super Computing Center[7]. Most experiments were performed with a 2.1 GHz Skylake-SP processor, 6TB RAM, and 18 cores. Other experiments were done on a Dell Precision T5600 workstation with 2 Intel Xeon E5-2630 processors (6 Core 2.2GHz) and 128GB RAM, running Linux Ubuntu 14.04 in a Docker container under Windows 7.

5.2 Benchmarks

Our experiments use two sets of benchmarks, one from the literature and another that we created.

Phase Ordering. This set comprises the 6 original programs from [Holder et al., 2017], and then we applied Tigress transformations by single type (abstract, control, data) on the 6 target programs one at a time, followed by application of all permutations of the obfuscation types (abstract, control, data), and finally, virtualization applied in combination with each permutation resulting in 21 different sequences of transformations. This resulted in 126 obfuscated programs plus the 6 original, for a total of 132 sample programs.

Tigress Pass/Code. This set includes 288 programs we created: 16 generated with the random program option from Tigress with a single password check ranging from 1 to 16 characters, 16 programs with single PIN code with size ranging from 1 to 16 digits,

---

[7] http://www.asc.edu

and 256 programs with both password and PIN ranging from 1 to 16 characters and digits each.

5.3 Effectiveness (RQ1)

Methodology. We repeat the experimental analysis of [Holder et al., 2017] and extend their results by analyzing the **Phase Ordering Benchmarks** using both KLEE and Angr. Our results produce different (longer) times than reported by Holder et al. (mainly due to a difference in Tigress version as we used version 2.2), but still confirm that no sequence of obfuscation types from Tigress could effectively prevent the recovery of the embedded program secret/property (password or PIN code). Most transformation sequences increase analysis time but some are ineffective, requiring roughly the same amount of analysis time on the unprotected original programs, and some even defective, actually reducing the analysis time. Full data can be found in the technical report and benchmarks (URL given Page 4). In summary, regarding our 132 test programs, KLEE and Angr finished analysis in all experiments with a max runtime of 9935 seconds (2.75 hours) for Angr and 42 seconds for KLEE. *In all cases, KLEE and Angr revealed the embedded secrets (password and PIN) of the sample program (132/132 attack successes for each tool).*
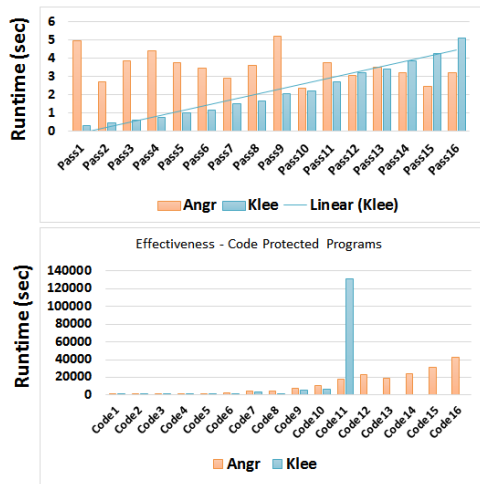

Figure 5: Tigress Pass/Code Only Benchmarks analysis

We also evaluate the *unobfuscated* programs in the **Tigress Pass/Code Benchmark** set using KLEE and Angr. Fig. 5 to 7 illustrate the overall results of the study. Tabular results of the data are provided in the separate technical report and benchmarks (mentioned in Page 4). We were able to retrieve the password from all single password programs (Pass1-Pass16) using both Angr and KLEE within 5 seconds. For single
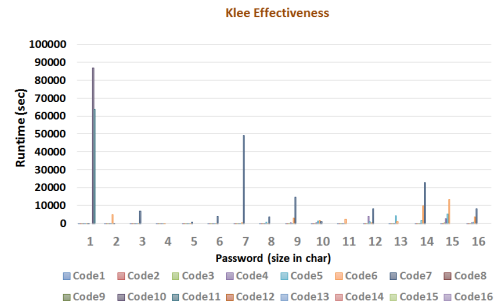
Figure 6: Tigress Pass/Code Combined Benchmarks analysis
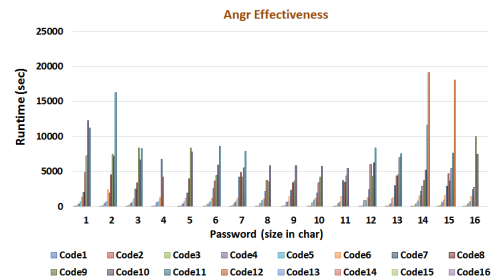
Figure 7: Tigress Pass/Code Combined Benchmarks analysis

PIN code programs, Angr is able to retrieve the code from all the programs (code1-code16) whereas KLEE only retrieves code from a portion (code1-code11). Angr takes up to 42950 seconds (11.9 hours) and KLEE. 131190 seconds (36.4 hours). On average, the mean time to crack for Angr was 3.55 sec for passwords and 11,908 sec for passcodes; for KLEE it was 2.17 sec for passwords and 13,485 sec for passcodes. The minimum time for Angr was 2.39 and 26.5 secs for passwords and codes; the min time for KLEE was 0.36 and 1.85 for passwords and codes. We allowed a timeout of 120 hours for both KLEE and Angr in failure cases.

For programs that had both password and PIN code, Fig. 6 and 7 shows results for KLEE and Angr analysis. Tabular results are provided in a separate technical report. We evaluated 288 unprotected samples in 576 experiments; for password and code only programs Angr leaked the password in 32 of 32 cases, KLEE leaked the password in 27 of 32 cases; for combined pwd/pin programs, KLEE leaked both pwd/pin in 120 of 256 cases and Angr leaked both pwd/pin in 169 of 256 cases. We allowed a timeout of 120 hours for both KLEE and Angr in failure cases. Angr had a maximum analysis time of 5.3 hours and KLEE had a maximum analysis time of 79.3 hours. *KLEE and Angr achieves high attack success rates (resp. 142/288 and 201/288).*

Results with SBHA. We apply SBHA to all programs in the **Phase Ordering Benchmark** set, on top of 12

Figure 8: SBHA Effectiveness against Angr/KLEE

out of each 22 obfuscated variants and each original program: we could not create the remaining variants as the obfuscation transformations generated by Tigress did not allow precise location of the point function. The 39 variants (13*3) were analyzed with both Angr and KLEE, *with a timeout of 120 hours for every sample file analyzed. In summary, adding SBHA on top of existing transformation sequences in the* Phase Ordering Benchmark *resulted in a complete defeat of the attacker tools Angr and KLEE (no attack success, 0/39 for both tool).* Similarly, for the **Tigress Pass/Code** benchmark set no analysis retrieved either the single password, single PIN code, or combined password/PIN from any of the SBHA variants, except for KLEE recovering the single digit PIN code from the PIN code-only set of SBHA transformed programs, as Fig. 8 illustrates. *In summary, we analyzed 288 SBHA-only protected programs with KLEE and Angr; KLEE and Angr were given a 120-hour timeout period; for both tool the attack succeeded only in 1/288 case, which was a single-password protected program with a 1 character password.*

RQ1 Conclusion. SBHA is extremely effective against DSE attacks. Indeed, both Angr and KLEE were unable to retrieve passwords from any but one (1/327) of the protected variants including the SBHA version of the original programs (despite a huge 120h time-out per sample), where these tools recover a significant part of the passwords without SBHA (KLEE: 181/327, Angr: 240/327).

5.4 Resilience (RQ2)

Methodology. We measure the strength of SBHA obfuscation transformations against simplification or deobfuscation in the following 3 manners.

Since Tigress programs are C source code, we used `GCC` optimization levels to gauge how much code is reduced by the compiler, as an indication of how much obfuscated code is reduced.

Assuming that an attacker could successfully recover the circuit netlist from the application binary code, we utilize two tools (ABC [Brayton and

Figure 9: Evaluation Case Study for Resilience

Mishchenko, 2010] and JDD[8]) to evaluate whether circuit-based analyzers can recover the embedded password from the circuit itself. Fig. 9 illustrates our evaluation framework for resilience.

SBHA can be used to target the path explosion weakness inherent in typical DSE engines. However advanced DSE somewhat counters that explosion by using path merging [Kuznetsov et al., 2012]. We approximate path merging attacks by evaluating SBHA C programs where Boolean operators (!, ||, &&) are used instead of conditional statements, as seen in the example code in Program 3.

Compiler Optimizations. We use GCC optimization levels (0, 1, 2, 3, s, fast) to estimate both resilience and overhead. An optimizing compiler like GCC can be seen as a basic adversarial attacker that would deobfuscate an obfuscated program by optimizing away the included protections. We transformed the original 6 Phase Order Benchmark programs with SBHA, compiled them under the 6 different GCC optimization levels, and then analyzed the 36 program using Angr. After 120 hours of computation for each variant, the tool still found no solution. We thus claim that SBHA abstractions are resilient to compiler optimizations since they do not improve analysis by an automated adversarial tool like Angr. *Additional details can be found the companion technical report.*

> Optimizations refactor logic circuit abstraction code into different instructions, but do not optimize netlists enough to thwart the protection.

Circuit Recovery Attacks. We now turn to evaluating the resilience of our circuit netlists against adversarial analysis (see Fig. 9 for a summary). We assume that our adversary has recovered the circuit netlist:

---

[8] https://bitbucket.org/vahidi/jdd/src/master/

can they retrieve the original point function from that representation? Since the hardware abstraction uses a circuit netlist to represent the conditional point function check, we assumed that an adversary would recover some form of the circuit netlist from the binary code. For analyzing resilience, we used ABC and JDD as two representative tools that can be used to recover the point (password) from the netlist once it is recovered. *We applied both anti-SAT and anti-BDD transformations* (as explained in Sec. 4.2) to representative SBHA point function netlists to characterize the success of attacks using these tools.

We first generated point-function circuits with input size as low as 8 bits (1 character) up to 14000 bits (1750 characters): ABC was able to return a PLA definition of every circuit between 1 and 35 seconds, where the PLA reveals the single input required to produce a 1 (or true) for the circuit. JDD was able to produce a ROBDD for every circuit as well for every point function circuit, and the ROBDD directly shows the point (sequence of inputs) that would produce the true result of the circuit.

We applied anti-SAT transformations to the Tigress Pass/Code benchmark circuits produced during SBHA transformation and evaluated those in the same manner. Our anti-SAT transformations caused ABC to fail to produce a PLA, even on the lowest sized point-function circuits (1 character or 8 bits). Likewise, the anti-BDD transformation caused JDD to fail to create a BDD (heap and stack overflows based on memory) for the same, including the lowest point-size function (1 character or 8 bits). Our analysis indicates that both of our anti-SAT and anti-BDD approaches provide high resilience against the two circuit based analysis tools that we studied.

> SBHA *equipped with anti-SAT and anti-BDD transformations* is resilient to circuit recovery attacks.


```
bool and(bool a, bool b) {
    if (a) {
        // 11
        if (b) {
            return true;
        }
        // 10
        else {
            return false;
        }
    }
    else {
        // 01
        // 00
        return false;
    }
}
```

```
bool and(bool a, bool b) {
    return a & b;
}
```
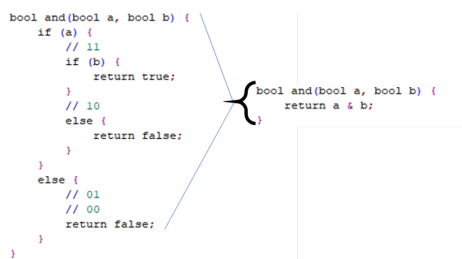
Figure 10: Path Merging Boolean Library

Path Merging Attacks. Fig. 10 shows snippets from the Boolean code library used to simulate a path merging attack, showing the reduced form of the

AND or OR functions in C code. By splitting operators and getting rid of IF statements, we are essentially getting rid of paths taken in the code, which is equivalent to what an optimizing DSE engine would do to counter path explosion. In this regard, we are experimenting with a worst case attack, where other forms of obfuscation of the Boolean library itself (many different alternate forms for each Boolean logic function in C code, obfuscating the Boolean data type itself, etc.) have failed and the reverse engineering has achieved path merging [Kuznetsov et al., 2012].

We first tested the 8 character password-only program that simulates a path-merging attack (Fig. 10) using KLEE and Angr. KLEE failed to return a password in 24 hours while Angr cracked the 8 character password in $\approx$ 1 hour computation time for the same sample. Nonetheless, 1 and 2 character password simplified SBHA C programs *with circuit-based countermeasures* for the point function resisted against KLEE and Angr runs during 24 hours (a descriptive figure for anti-SAT blocks is provided in the technical report). This result illustrates that a worst-case path-merging attack requires additional protective measures, which, when added, can effectively counter the initial attacker.

> Path-merging can somewhat threaten SBHA, but this can be recovered by adding (HW-inspired) anti-SAT and anti-BDD techniques.

RQ2 Conclusion. SBHA offers high resilience to compiler optimizations, and high resilience to circuit recovery attacks and path-merging techniques when equipped with HW-inspired protections.

5.5 Stealth (RQ3)

Methodology. Our approach is two-fold: 1) we use the MOSS program similarity checker [Schleimer et al., 2003] to see the similarity between the original programs and obfuscated variants, and 2) we compute op-code distributions of the original versus obfuscated programs. We use programs in the Phase Ordering Benchmark set to analyze stealth.

Results. Fig. 11 provides an appropriate view on how well SBHA code blends into existing code. SBHA is not stealthy because circuit abstractions are unusual w.r.t. other surrounding code. Our comparison of original point function programs to SBHA transformed executables show that certain opcodes provide a signature for the presence of SBHA (*mov, movzx* for example). Applying SBHA consistently across other
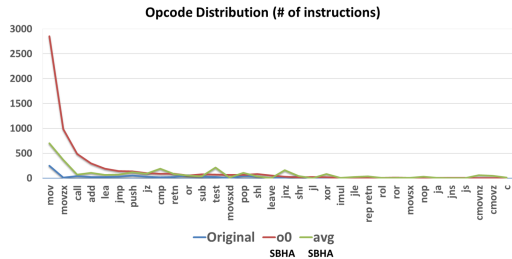
Figure 11: Comparative Opcode Analysis

code constructs than point functions in a target program would certainly help in hiding its presence to protect the identified program assets.

RQ3 Conclusion. SBHA alone does not have good stealth, because the presence of a custom Boolean library and an embedded circuit definition are tell-tale indicators of SBHA use. Using SBHA in conjunction with other traditional software-based obfuscating transformations can normalize the presence of SBHA code. Likewise, using SBHA in multiple locations throughout the code would at least normalize the presence of SBHA-based statements, but requires transformation of general program code sequences.

5.6 Cost (RQ4)

Methodology. We calculate cost in terms of source lines of code (LOC), memory overhead, and execution time overhead of the SBHA obfuscated programs. We used the Phase Ordering Benchmark set to estimate cost.

Table 1: Overhead Analysis: Original vs SBHA

	Original			SBHA					
	LOC	Size	Runtime	LOC	xLOC	Size	xSize	Runtime	Delta
Pass6	174	12816	0.084	709	3.07	17704	38%	0.084	0.00
Pass9	113	8720	0.081	750	5.64	17704	103%	0.084	0.00
Pass 12	195	12816	0.083	938	3.81	21800	70%	0.082	0.00
Pass3Code2	256	12824	0.001	773	2.02	17784	39%	0.001	0.00
Pass3Code3	186	12824	0.091	731	2.93	17784	39%	0.083	0.01
Pass4Code4	172	12824	0.091	789	3.59	17784	39%	0.082	0.01
				Avg xLOC:	3.51	Avg Size:	55%		

Results. Table 1 provides a summary of the key metrics (LOC, size, runtime) between original programs and SBHA variants. Fig. 12 shows further analysis of with executable size and LOC with Tigress obfuscations added. Virtualization techniques are usually very expensive, both in terms of code size increase and runtime penalty [Ollivier et al., 2019b]. SBHA does not suffer as much from these known drawbacks. Runtime difference of executables is low: we record a maximum of 0.01 seconds (10%) between original and SBHA variants. Since the programs evaluated were *only* authentication based, the overhead represents a pure assessment of SBHA. Source lines of

code (SLOC) provide a measure of overall code increase, with $3.5\times$ average overhead for SBHA transformation (min $2.02\times$, max $5.64x\times$). SLOC and executable size have a mostly linear relationship. Executable size at the unoptimized compilation level increases $\approx 55\%$ onaverage, with a minimum of 38% and maximum of 103%.
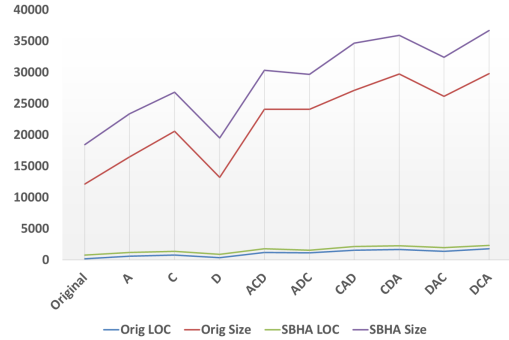

Figure 12: Code Size Analysis With Tigress Obfuscation

Fig. 12 shows extended analysis of SBHA variants that were further obfuscated using Tigress transformations (A, C, D, ACD, ADC, CAD, CDA, DAC, DCA). In these cases, we chose to do SBHA transformation after Tigress was applied, and therefore virtualized versions were not considered. Overall, SBHA provides a predictable constant increase in LOC and binary executable size (in bytes). The results illustrate the varying overhead of various combinations of Tigress obfuscations, but a constant overhead from SBHA included.

RQ4 Conclusion. SBHA is a low-cost transformation for point-function constructs, even more so for a virtualization techniques. Based on observed data, we surmise that SBHA is a cheap (linear) transformation, based on Collberg et al. scale [Collberg and Nagra, 2009].

6 Related work

Protections against dynamic symbolic execution (DSE) have been well studied in the last decade, as well as proposed techniques to protect point function constructs in particular. [Schrittwieser et al., 2016] provide a recent exhaustive survey that highlights the unreasonable efficiency of DSE-based deobfuscation [Bardin et al., 2017, Yadegari et al., 2015, Coogan et al., 2011, Salwan et al., 2018]. [Banescu et al., 2016, Banescu et al., 2015] provided one of the first comprehensive evaluations of DSE tools (KLEE, Angr) against variants of password programs using

obfuscators such as Tigress and ObfuscatorLLVM. Their evaluation was seminal in terms of highlighting the great weakness of standard protections against DSE. [Manikyam et al., 2016] performed a similar study on commercial obfuscators such as Themida, Code Virtualizer, and VMProtect. The impact on symbolic deobfuscation through the complexification of constraints has also been studied by [Banescu et al., 2017], where they showed machine learning could be used to predict effectiveness of obfuscating transformation in terms of increasing runtime of DSE-based attacks. More recently, [Ollivier et al., 2019b] provided successful results for the so-called class of path-oriented protections that target the weakest spot of DSE, namely path exploration. Their target of interest also included password-based programs similar to our study in this paper. The same authors also propose a survey of anti-DSE protections [Ollivier et al., 2019a]. Before this, several obfuscating transforms were proposed by [Biondi et al., 2017] and [Eyrolles et al., 2016] based on Mixed Boolean Arithmetic expressions [Zhou et al., 2007] to protect point-functions. [Bruni et al., 2018] also proposed a mathematically proven obfuscation against Abstract Model Checking attacks.

[Lan et al., 2018] is the closest related work in software protection: they evaluate an obfuscator that replaces sensitive conditional instructions with semantically equivalent function calls with a more complicated execution model. This work showed similar success with defeating symbolic analysis attacks using KLEE. SBHA can be seen as path-oriented protection [Ollivier et al., 2019b], enjoying the (strong) so-called property of "Single-Value Path" defined by Ollivier et al. Intuitively, SVP protections turn DSE search into mere fuzzing. Only a very few tractable SVP path-oriented protection schemes are known, making SBHA a new defense. Also, contrary to the work by [Ollivier et al., 2019b], SBHA opens the way to a whole class of such schemes (the principle is not limited to point functions). SBHA presents program abstractions that are fundamentally abnormal for a software-minded human attacker, more so than standard path-oriented protections present. In addition, it now provides potential reuse of circuit-based protections in the context of software.

7 Conclusion and Future Work

We introduce SBHA-based software transformation as a completely new approach to software protection. Our initial study on protecting point-function programs, particularly against DSE-based attacks, shows great promise. Experimental results point to the effectiveness of SBHA in defeating DSE, with relatively high resilience and low overhead. SBHA has expectedly low stealth in terms of blending into surrounding code, yet we anticipate that application to general code would help to normalize its presence throughout a code base. Still, our work demonstrate the potential of the technique. Interestingly, SBHA provides a way to unify two normally disparate worlds or research: software and hardware protection. Whereas obfuscation is generally frowned upon in the hardware world because of constrained power, space, or timing demands, the equivalent representation in software may not be at all. Future work include expanded study of SBHA in larger program contexts and subjective study of human-based reverse engineering limitations when encountering SBHA variation.

REFERENCES

Balakrishnan, G. and Reps, T. W. (2010). WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6).

Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., and Pretschner, A. (2016). Code obfuscation against symbolic execution attacks. In *ACSAC'16*.

Banescu, S., Collberg, C., and Pretschner, A. (2017). Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *USENIX SEC'17*.

Banescu, S., Ochoa, M., and Pretschner, A. (2015). A framework for measuring software obfuscation resilience against automated attacks. In *SPRO'15*.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., et al. (2012). On the (im)possibility of obfuscating programs. *J. ACM*, 59(2).

Bardin, S., David, R., and Marion, J. (2017). Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes. In *S&P '17*.

Basile, C., Canavese, D., Regano, L., Falcarin, P., and Sutter, B. D. (2019). A meta-model for software protections and reverse engineering attacks. *Journal of Systems and Software*, 150.

Beyer, D. and Stahlbauer, A. (2014). Bdd-based software verification. *Int. J. Softw. Tools Technol. Transf.*, 16(5).

Biondi, F., Josse, S., Legay, A., and Sirvent, T. (2017). Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70.

Brayton, R. K. and Mishchenko, A. (2010). ABC: an academic industrial-strength verification tool. In *CAV'10*.

Bruni, R., Giacobazzi, R., and Gori, R. (2018). Code obfuscation against abstract model checking attacks. In *Verification, Model Checking, and Abstract Interpretation*. Springer.

Bryant, R. E. (1991). On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2).

Cadar, C. and Sen, K. (2013). Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2).

Ceccato, M., Tonella, P., Basile, C., Coppens, B., De Sutter, B., Falcarin, P., and Torchiano, M. (2017). How professional hackers understand protected code while performing attack tasks. In *ICPC'17*.

Cheng, X., Lin, Y., Gao, D., and Jia, C. (2019). DynOpVm: VM-Based Software Obfuscation with Dynamic Opcode Mapping. In *Applied Cryptography and Network Security*.

Collberg, C. and Nagra, J. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical report, Univ. of Auckland.

Collberg, C. S. and Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Trans. on Soft. Eng.*, 28(8).

Coogan, K., Lu, G., and Debray, S. (2011). Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *CCS'11*.

De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education.

Eyrolles, N., Goubin, L., and Videau, M. (2016). Defeating MBA-Based Obfuscation. In *SPRO'16*.

Falcarin, P., Collberg, C., Atallah, M., and Jakubowski, M. (2011). Guest editors' introduction: Software protection. *IEEE Software*, 28(2).

Hansen, M. C., Yalcin, H., and Hayes, J. P. (1999). Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering. *IEEE Des. Test*, 16(3).

Holder, W., McDonald, J. T., and Andel, T. R. (2017). Evaluating optimal phase ordering in obfuscation executives. In *7th SSPREW*.

Junod, P., Rinaldini, J., Wehrli, J., and Michielin, J. (2015). Obfuscator-LLVM – Software Protection for the Masses. In *SPRO'15*.

Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. (2012). Efficient state merging in symbolic execution. In *PLDI'12*.

Lan, P., Wang, P., Wang, S., and Wu, D. (2018). Lambda obfuscation. In *SecureComm'17*.

Manikyam, R., McDonald, J. T., Mahoney, W. R., Andel, T. R., and Russ, S. H. (2016). Comparing the effectiveness of commercial obfuscators against mate attacks. In *SSPREW'16*.

Ollivier, M., Bardin, S., Bonichon, R., and Marion, J. (2019a). Obfuscation: where are we in anti-dse protections (a first attempt). In *SSPREW '19*.

Ollivier, M., Bardin, S., Bonichon, R., and Marion, J.-Y. (2019b). How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *ACSAC'19*.

Rival, X. and Yi, K. (2020). *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press.

Salwan, J., Bardin, S., and Potet, M.-L. (2018). Symbolic deobfuscation: From virtualized code back to the original. In *DIMVA'18*.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *ICMD'03*.

Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., and Weippl, E. (2016). Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1).

Subramanyan, P., Ray, S., and Malik, S. (2015). Evaluating the security of logic encryption algorithms. In *HOST'15*.

Vahid, F. (2007). It's time to stop calling circuits "hardware". *Computer*, 40(9).

Woelfel, P. (2005). Bounds on the obdd-size of integer multiplication via universal hashing. *Journal of Computer and System Sciences*, 71(4).

Yadegari, B., Johannesmeyer, B., Whitely, B., and Debray, S. (2015). A generic approach to automatic deobfuscation of executable code. In *S&P'15*.

Zhou, Y., Main, A., Gu, Y. X., and Johnson, H. (2007). Information hiding in software with mixed boolean-arithmetic transforms. In *WISA'07*.