IR Preprocessing for Deep Binary Analysis

Julian Kranz, Bogdan Mihaila, Holger Siegel, and Axel Simon

***Abstract*—Verifying the absence of vulnerabilities in executable programs (binaries) requires a precise static analysis that combines several abstract domains to infer targets of indirect jumps, bounds for array accesses, precise points-to information, etc. A particular challenge for such a complex analysis is the size of the input: Each source code line translates to several machine instructions; the semantics of a machine instruction, in turn, is expressed using several operations in an intermediate representation (IR), so that a code increase by a factor of 10 is not unusual. We show that applying a few classic program optimizations to this IR leads to a considerable size reduction and, thereby, to a faster analysis. Moreover, fusing several IR instructions also recovers some of the original high-level computation. As a consequence, fewer domains need to be implemented in the static analyzer to achieve the required level of precision. This work presents the front-end that generates and optimizes the IR. Additionally, we present a static analysis tool based on our IR that is used to measure the size and precision improvements.**

I. Introduction

The analysis of binary code is necessary if the source code of programs or third-party libraries is not fully available or if it contains inline assembler. Many applications of binary analysis exist, including the understanding of malware [14], [17], comparison of binaries in virus scanners [7], [18], creation and understanding of exploits [2] and many more. A prerequisite for many analyses is the reconstruction of the control-flow graph (CFG) that, due to the use of computed jumps and calls, requires a semantic analysis of the program [1]. The challenge here is one of scalability: on average, every line of C code translates to two assembler instructions [10]. Moreover, every assembler instruction $i \in \mathbb{I}$ usually has several effects that need to be considered in an analysis. For example, subtracting two registers will also set several processor flags that allow checking for overflow, carry, etc. These effects are translated into a sequence $p_1 \cdot \ldots \cdot p_n \in \mathbb{P}^*$ of primitive operations $\mathbb{P}$. Let $sem : \mathbb{I} \to \mathbb{P}^*$ implement this translation. The benefit of translating the native processor instructions into a small set of primitives is that a static analysis has to model the semantics for only $|\mathbb{P}|$ primitives instead of $|\mathbb{I}|$ instructions. In case we use the set of primitives of the RReil language [8] for the representation of Intel x86 instructions, this amounts to a reduction from $|\mathbb{I}| = 898$ to $|\mathbb{P}| = 28$. Since a precise static analysis uses a product of several abstract domains [4], say $D_1 \times \ldots \times D_k$ (such as value-sets, intervals, equalities [6], and congruences [5]), implementing $k|\mathbb{P}|$ instead of $k|\mathbb{I}|$ semantic functions (transformers) is highly desirable. We therefore implement the semantics as $tr_D : \mathbb{P} \to (D \to D)$ that describes how the primitives $\mathbb{P}$ affect the domain D. Define the lifting to a sequence of primitives as $\overline{tr_D}(p_1 \cdot \ldots \cdot p_n) = tr_D(p_n) \circ \ldots \circ tr_D(p_1)$ where $g \circ f = \lambda x \,.\, g(f(x))$.

Consider the evaluation of a basic block consisting of the instruction sequence $\bar{i} = i_1 \cdot \ldots \cdot i_n \in \mathbb{I}^*$ on a domain D by defining $bblock_D(\bar{i}) : D \to D$ as follows:

$$\begin{aligned} bblock_D(\bar{i}) &= \overline{tr_D}(sem(i_n)) \circ \ldots \circ \overline{tr_D}(sem(i_1)) \quad (1) \\ &= \overline{tr_D}(sem(i_1) \cdot \ldots \cdot sem(i_n)) \end{aligned}$$

In general, a native instruction $i \in \mathbb{I}$ translates to several primitives $sem(i)$. For instance, if we count each RReil statement as a primitive, then $|sem(i)| = 5.1$ on average [8], so that each C source code line turns to about 10 primitives. Given this growth, it is natural to perform some optimizations. A recent approach to defining efficient abstract transformers, called TSL [9], is to generate an optimized domain transformer for each native instruction up front. For this, instead of (1), the developers of TSL evaluate a basic block as follows:

$$bblock_D^{TSL}(\bar{i}) = opt_D(\overline{tr_D}(sem(i_n))) \circ \ldots \circ opt_D(\overline{tr_D}(sem(i_1)))$$

The novelty behind TSL is to emit C++ code for each $\overline{tr_D}(sem(i))$ where $i \in \mathbb{I}$ so that $opt_{D \to D}$ is given by standard compiler optimizations. Their intent is to perform this code generation for each domain D in the analysis. In this work, we propose to optimize a sequence of primitives. In particular, we apply $opt_{\mathbb{P}} : \mathbb{P}^* \to \mathbb{P}^*$ to basic blocks, so that our analysis evaluates the following:

$$bblock_D^{IR}(\bar{i}) = \overline{tr_D}(opt_{\mathbb{P}}(sem(i_1) \cdot \ldots \cdot sem(i_n)))$$

In fact, our $opt_{\mathbb{P}}$ also incorporates information from successor basic blocks (when known) in order to apply more aggressive optimizations. We observe that these optimizations can remove certain low-level artifacts that otherwise require the use of specialized abstract domains. In particular, certain domains $D_1 \ldots D_i$ become expendable, in that $bblock^{IR}_{D_1 \times \ldots \times D_i \times \ldots D_k}$ can be replaced by a more efficient analysis $bblock^{IR}_{D_{i+1} \times \ldots D_k}$ while yielding the same results for all practical purposes. Our contribution can therefore be summarized as a win-win solution:

- Applying $opt_{\mathbb{P}}$ on a basic block nearly halves the number of primitives per instruction from $|sem(i)| \approx 5.1$ to ~ 2.6. We present the GDSL toolkit in which $opt_{\mathbb{P}}$ is implemented once and applied to many architectures. The reduction in input size leads to a significant speedup of our static analyzer as shown by our measurements.
- We show how the optimized primitives provide a more high-level view of the input program for which a cheaper analysis with fewer domains suffices. We illustrate this by measuring the precision of our analyzer with and without certain IR optimizations.

The remainder of the paper is organized as follows. The next section illustrates the effect of $opt_{\mathbb{P}}$. Section III presents a binary analyzer targeted at control flow graph reconstruction which uses a stack of abstract domains. The performance comparison is given in Sect. IV before Sect. VI concludes.

II. Optimizing the Intermediate Representation

We illustrate the effect of our optimizations $opt_{\mathbb{P}}$ on the IR generated for the following C code:

```
1  int a = ...; int b = ...;
2  if(a < b) return h(a); else return g(x);
```

Compiling the code in line 3 results in the x86-64 code shown in the left listing below. The function calls in both branches are tail calls and thus turned into jumps. On the right, the non-optimized RReil code for the first two instructions is given:

```
3     CMP EBP, EAX
4     JGE then
5   else:
6     MOV EDI, EBP
7     ADD RSP, 8
8     POP RBX
9     POP RBP
10    JMP g
11  then:
12    MOV EDI, EBX
13    ADD RSP, 8
14    POP RBX
15    POP RBP
16    JMP h
```

```
17  IP =:64 (IP + 2)
18  T0 =:32 (BP - A)
19  FLAGS.7 =:1 T0 <s:32 0
20  FLAGS.6 =:1 BP ==:32 A
21  FLAGS =:1 BP <u:32 A
22  LEU =:1 BP <=u:32 A
23  LTS =:1 BP <s:32 A
24  FLAGS.11 =:1
25    LTS ^ FLAGS.7
26  LES =:1 BP <=s:32 A
27  FLAGS.4 =:1 BP <u:4 A
28  ... 3 assignments
29    omitted ...
30  IP =:64 (IP + 2)
31  T5 =:1 LTS ^ 1
32  cbranch T5 then else
```

The RReil code consists mostly of assignments, where `lhs =:`s `rhs` assigns and `lhs <u:`s `rhs` compares s bits of `rhs` and `lhs` where `<u` is an unsigned comparison. The statement **`cbranch`** `c t e` jumps to t if the one-bit expression c is one and to e otherwise. In the IR code, all effects of the comparison, especially those on the `FLAGS` register, are explicit. Note that many results are never used since only `LTS` flows into the **`cbranch`** statement and since the x86 `ADD` instructions in lines 7 and 13 overwrite all bits of the `FLAGS` register. In a first pass, we perform a liveness analysis and eliminate dead assignments. This results in the following code:

```
33  IP =:64 (IP + 2)
34  LTS =:1 BP <s:32 A
35  IP =:64 (IP + 2)
36  T5 =:1 LTS ^ 1
37  cbranch T5 then else
```

In another pass, we delay and forward substitute simple expressions [12, Chap. 1.15] where possible. This allows us to combine the two assignments to the program counter. Moreover, we apply simple reductions that, for instance, match the negation in line 36 and that transform the comparison in line 34 to `A <=s:32 BP`. Without this transformation, an additional domain is required in the static analysis tool to recover the relation between the flag `T5` and the two registers `BP` and `A`. The final RReil code is minimal for our example:

```
33  IP =:64 (IP + 4)
34  cbranch A <=s:32 BP then else
```

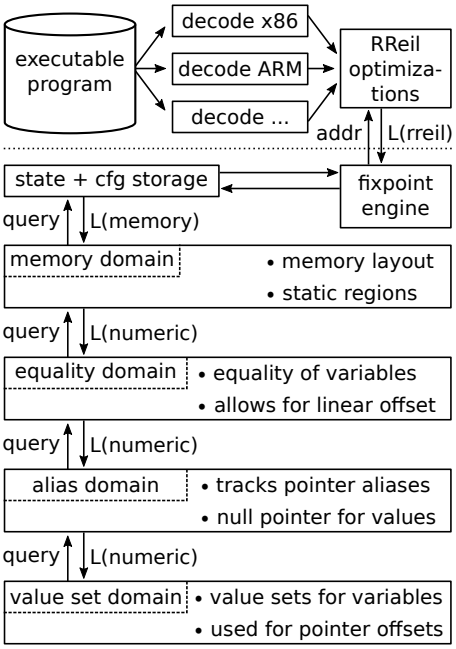

Figure 1. Analyzer structure.

In summary, $opt_{\mathbb{P}}$ reduces the size of the RReil code considerably and recovers high-level information of the computation of branch conditions. This enhances the readability of the IR, and, more importantly, increases the performance and precision of analysis tools.

III. Experiment Fixture: A Static Analyzer for CFG Reconstruction

We give a brief overview of our analyzer for executables in order to provide the background for understanding our experimental data. Figure 1 presents the overall structure of the analyzer. The decoding, translation into RReil, and optimization shown above the dotted line is packaged into the GDSL toolkit [8], [13], [16] and can be used independently of any analysis. The part below the dotted line is an analysis geared towards the recovery of indirect jumps and indirect calls. Together, this information provides the intra- and inter-procedural control-flow graph (CFG) of the program.

The analysis is interprocedural in that it computes summaries for functions. A summary relates the input of a function (i.e. the memory that is read) to its output (i.e. the memory written). Our summaries can express linear relations between the input and output and, thus, enable us to express the flow and typical modifications of pointers. For each program variable x, the abstract domain state S_f contains two so-called domain variables $x_{\texttt{in}}, x_{\texttt{out}}$ that express the input/output relation of x. For instance, $x_{\texttt{out}} = x_{\texttt{in}} + 1$ states that the program variable x has been incremented by one.

The analysis is bootstrapped by computing a summary $S_f \in D$ for each trivially identified function f, e.g. those found in direct calls or declared in the ELF header. Indirectly called functions are only discovered during the analysis. Specifically, when encountering an indirect call `call x` in a function f and x is represented by $x_{\texttt{out}} = x_{\texttt{in}}$, the known callers of f are

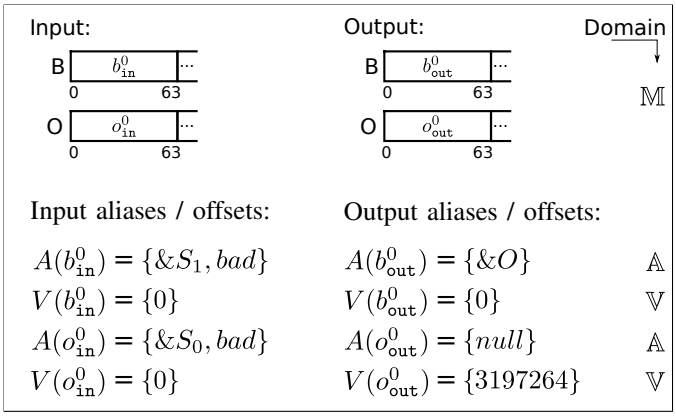

Figure 2. Analysis state after object construction in line 38

used to find values for x_{in}, and thus x_{out}. The computation of the summary S_f is continued using the function pointers in x_{out} provided by the callers. In particular, a summary may be recomputed when the known callers change, namely, if the new callers provide different input function pointer sets. Moreover, callers are re-evaluated whenever the summary of the called function changes. Note that this means that information is propagated in two ways: the effect of the summary is applied to the state at each call site while additional function pointers in the caller state lead to a re-evaluation of the summary.

In order to obtain precise jump and call targets, the analyzer state is composed of several domains (the memory, equality, alias, and value set domain) that are organized in a hierarchy. The remaining section gives an example of how the analyzer uses these domains to resolve a branch.

Consider the following C++ program:

```
33  struct Base { virtual void f() = 0; };
34  struct Sub : public Base
35    { void f() { ... } };
36
37  int foo(Base *B) {
38    B = new Sub();
39    B->f();
40    return 0;
41  }
```

The program creates an object of class `Sub` in line 38 and calls the virtual function `f` on this object in line 39. The call to the constructor of `Sub` fills in the `vtable` pointer of the newly created object. The relevant part of the domain state after evaluating the RReil code of line 38 is shown in Fig. 2.

In the figure, the *memory domain* $\mathbb{M}$ tracks two regions - one for the C++ variable[1] B and one for the object O of dynamic type `Sub`. The memory domain tracks a set of fields analogous to the fields of a C `struct`. For instance, the field b_{in} represents bits 0 through 63 of the input of region B. Each memory region has an input and an output variant. In the example, their fields are named b_{in}, b_{out} and o^0_{in}, o^0_{out}.

The link between the pointer stored in $b_{\text{in}}/b_{\text{out}}$ and the pointed-to memory is managed by the domains below the memory domain in Fig. 1. For simplicity, Fig. 2 only shows the alias and value set domain states. The alias domain state

[1]On the binary level there are only registers and no C++ variables. For simplicity, we assume to have a one-to-one mapping between C++ variables and registers in the example.

	RReil statements			cond. inlined
	no opt.	all opt.	red. by	by fsubst.
`echo`	18k	8778	51%	97%
`sleep`	18k	9048	50%	95%
`cat`	39k	20k	49%	95%
`gdsl-arm`	445k	210k	53%	89%

Figure 3. RReil code size metrics

	code coverage		resolved branches	
	with fsubst.	without fsubst.	with fsubst.	without fsubst.
`echo`	63%	53%	2 (40%)	0 (0%)
`sleep`	67%	58%	2 (33%)	0 (0%)
`cat`	78%	74%	4 (50%)	1 (13%)
`gdsl-arm`	46%	33%	64 (41%)	5 (4%)

Figure 4. Analyzer precision

$A \in \mathbb{A}$ states that the C++ variable B is an alias of some valid symbolic pointer $\&S_1$ or an invalid pointer bad at the beginning of the function (as shown in the input alias set), but has been set to alias $\&O$ by the call to the constructor (as shown in the output alias set). The value set domain state $V \in \mathbb{V}$ is responsible for storing the pointer offset.

When calling the virtual method `f` in line 39, o^0_{out} contains the address of the `vtable`. The constructor has initialized o^0_{out} with a constant 3197264 which our analysis represents as an alias with $null$ with an offset of 3197264. While loading the address of `f`, this value is dereferenced. The analyzer treats accesses to constant addresses by mapping them to constant data sections of the ELF file and is thereby able to extract the function pointer in the `vtable` of O.

The whole analysis is driven by a fixpoint engine that translates the x86-64 code at a given address into optimized RReil and computes the state at the next address. This concludes the overview of our analyzer for executables which is open source and available at https://versioncontrolseidl.in.tum.de/backjmp/summy.

IV. Experimental Work

We tested optimization $opt_{\mathbb{P}}$ on binaries that are common to current Linux installations and also on the GDSL library itself (the GDSL library makes extensive use of indirect jumps through switch tables). Figure 3 contains statistics regarding the translation to RReil and the impact of our optimizations. The binaries are taken from an Ubuntu 16.04 installation. In the table, the second column contains the amount of statements generated for the binary with all optimizations turned off. As columns three and four show, turning on optimizations results in a code size reduction of up to 53% percent. As can be seen in the last column of the table, we are able to forward substitute almost every condition.

Turning optimizations on and off has an effect on the performance and the precision of our analysis tool. Figure 4 shows how the analyzer precision is affected by the forward expression substitution. Here, columns one and two contain the fraction of the binary reached by the analyzer with and

	all opt time	no liveness time	liveness speedup
echo	76s	185s	2.4x
sleep	159s	347s	2.2x
cat	1441s	3212s	2.2x
gdsl-arm	512s	685s	1.3x

Figure 5. Analyzer performance and speedup

without the optimization, respectively. As the analyzer only processes functions of the binary to which it has found a call, the coverage depends on the amount of branch targets recovered as shown in the *branches* columns. Here, we count all branches resolved by the analyzer, i.e. the branches which are either known to be unreachable or have at least one valid target inside the `.text` section of the binary. Figure 5 demonstrates that enabling optimizations not only increases the precision, but also the speed of the analyzer. Here, we only consider turning the liveness optimization on and off since this does not affect the analysis result and, thus, allows for a comparison of the running time. In the figure, the second and third column show the time required with and without the liveness optimization, respectively. The resulting speedup can be found in the last column.

Figure 3 shows that around 96 to 100 percent of the branch conditions can be forward substituted. An inspection of the optimized RReil code revealed two typical cases where a condition cannot be forward substituted into the cbranch instruction. The first case occurs when it is assigned to a flag that is used in a successor basic block. Since the next basic block might have other incoming edges, we cannot safely make assumptions about the content of flag variables in this situation. The second case arises when registers mentioned in a conditional are overwritten before the `cbranch` instruction. As an example, consider the following RReil code that compares two registers `B` and `C`, stores the outcome of the comparison in a flag register `F`, and then updates register `B` before a conditional jump is performed depending on the value of `F`:

```
F =:1 B <s:64 C;     B =:64 (B - C);
cbranch F label_true label_false
```

Here, inlining the conditional expression `B <s:64 C` for `F` is not possible, as the register `B` in the conditional expression it refers to is overwritten before the conditional jump is encountered. This situation can arise for compiled machine code since program variables are mapped onto a limited number of CPU registers. It is future work to find ways to alleviate this situation. One way that seems promising to us is to introduce different registers for different *versions* of register `B`, similar to the calculation of the *static single assignment* form.

In summary, due to the IR optimizations we are able to take branch conditions into account during our analysis, thus allowing us to achieve high precision without the need for an extra domain. Moreover, the analysis is faster when enabling optimizations due to the reduced code size.

V. Related Work

The approach of translating machine code into some kind of intermediate representation before running binary analysis is pervasive in the field of binary analysis. In contrast to other implementations, however, GDSL puts special emphasis on a clear separation of a translation and an optimization phase that processes the IR before passing it to an analysis.

A well-known intermediate representation is the VEX IR that is developed by the Valgrind project [11]. One notable difference between VEX and RReil is that VEX delays flag computations to where flags are actually needed. For example, an x86 `ADD` instruction sets an IR-specific variable expressing that the last operation was an addition. Additionally, special variables are used to track the values of the operands. If later a flag is read by a conditional branch, it is explicitly computed from this meta data. The VEX IR is used as intermediate representation by the Angr [15] binary analysis framework. Angr also applies optimizations like a liveness analysis to its IR during lifting. However there are no options to directly control the extend of the optimization. Tests show that the IR generated by GDSL has around 30% less statements and is, thus, more compact.

The BAP framework [3] uses its own IR called BIR, much like the GDSL toolkit. The `bap` tool does not apply a liveness optimization to the IR. However, since the IR is able to express more complex expressions it uses fewer instructions than RReil. In particular, BIR allows a right hand side expression to contain multiple shift operators, bitwise operations, and let bindings. This is not the case for RReil since its primary design goal was to keep the analysis transformers as simple as possible. As an example, consider the computation of the parity flag in BIR:

```
000000aa: PF := ~(low:1[let v202 =
  (RBX >> 0x4:64) ^ RBX in let v202 =
    (v202 >> 0x2:64) ^ v202 in
      (v202 >> 0x1:64) ^ v202])
```

The BIR IR only needs a single instruction here. RReil, on the other hand, uses three distinct assignment statements:

```
T9 =:4 T5 ^ T5.4
T9 =:2 T9 ^ T9.2
FLAGS.2 =:1 T9 ^ T9.1
```

In summary, a comparison between RReil and BIR cannot rely on the number of statements alone. Instead, we compared the amount of more complex operations in BIR to the RReil statement count. It turned out that the optimized RReil code is up to 2 times smaller according to this metric.

VI. Conclusion

We presented two tools: a front-end to decode, translate and optimize basic blocks in executables and a static analyzer that performs a fixpoint computation using a set of domains. We have argued that optimizations at the intermediate representation can lead to a drastic reduction in analysis times and allow to simplify the static analyzer. After applying the presented optimizations, our intermediate representation RReil is more compact than the IRs of other well-known analysis tools.

REFERENCES

[1] G. Balakrishnan, G. Grurian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 – A Platform for Analyzing x86 Executables. In Compiler Construction, volume 3443 of LNCS, pages 250–254, Edinburgh, Scotland, April 2005. Springer. Tool-Demonstration Paper.

[2] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In 2008 IEEE Symposium on Security and Privacy (sp 2008), pages 143–157, May 2008.

[3] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform, pages 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[4] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ Analyzer, pages 21–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[5] P. Granger. Static Analyses of Congruence Properties on Rational Numbers (Extended Abstract). In P. Van Hentenryck, editor, Static Analysis Symposium, pages 278–292, Paris, France, September 1997. Springer.

[6] M. Karr. On affine relationships among variables of a program. Acta Informatica, 6(2):133–151, 1976.

[7] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. Journal in Computer Virology, 7(4):233–245, 2011.

[8] J. Kranz, A. Sepp, and A. Simon. GDSL: A Universal Toolkit for Giving Semantics to Machine Language. In C. Shan, editor, Asian Symposium on Programming Languages and Systems, Melbourne, Australia, December 2013. Springer.

[9] J. Lim and T. Reps. A System for Generating Static Analyzers for Machine Instructions. In L. Hendren, editor, Compiler Construction, volume 4959 of LNCS, pages 36–52. Springer, 2008.

[10] Henrik Paul on StackOverflow. How Many ASM-Instructions per C-Instruction? http://stackoverflow.com/questions/331427, 2009.

[11] Valgrind Project. Valgrind - A GPL'd System for Debugging and Profiling Linux Programs. http://www.valgrind.org/, 2016.

[12] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. Compiler Design - Analysis and Transformation. Springer, 2012.

[13] A. Sepp, J. Kranz, and A. Simon. GDSL: A Generic Decoder Specification Language for Interpreting Machine Language. In Tools for Automatic Program Analysis, ENTCS, Deauville, France, September 2012. Springer.

[14] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A Framework for Enabling Static Malware Analysis, pages 481–500. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[15] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In IEEE Symposium on Security and Privacy, 2016.

[16] A. Simon and J. Kranz. The GDSL toolkit: Generating Frontends for the Analysis of Machine Code. In Program Protection and Reverse Engineering Workshop, San Diego, California, USA, January 2014. ACM.

[17] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis, pages 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[18] Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, pages 411–420, Dec 2007.