Schadenfreude: Resurrection

Addison Crump
Department of Computer Science
Texas A&M University
College Station, Texas
addisoncrump@tamu.edu

***Abstract*—Ghidra's inherent support for many different CPU architectures, executable formats, and user-developed plugins and softwares lends itself to be very strong in the inspection of a wide variety of software. While many plugins for Ghidra exist to enhance manual software reverse engineering, few exist to assist in the automation of such analysis. Where Schadenfreude first failed to meet the generalised needs for automation, the "Resurrection" branch promises to improve by integrating more generalised program analysis techniques as API. While the software is not yet suitable for general use, it demonstrates powerful functionality across programs whose control flow graph is acyclic. In future iterations, not only will these building blocks provide future researchers with the means to use these techniques in their own contexts, but also will improve the efficacy and scope of the existing analyses provided with the first iteration of Schadenfreude.**

I. Introduction

The open source software reverse engineering and vulnerability research communities are strong, but limited; with few automation assistants being published, a majority of this research is conducted manually or with arcane and highly specialised software which has been abused, misused, or modified to match the needs of the user.

With the release of the National Security Agency's software reverse engineering suite Ghidra [1] in 2019, a toolkit became available to the open source community that was well-reviewed and developed by a well-known organisation. This had obvious advantages: Ghidra is free (which supports smaller organisations and developers), it is open source (meaning the public can review it for errors), and can be extended rather simply. The way in which it was developed allowed users to develop extensions to support both automated analysis and manual review, and, as a key feature, provides an intermediary representation language which is consistent between CPU architectures and allows for more generalised program analysis.

Unfortunately, the software as released is mostly oriented towards manual review. Software can be developed to extend the automated analysis, but is mostly oriented towards control flow and data flow analysis with no awareness of contraints on the data or means by which to notate them. Viewing the source code and behaviour of the internal API, it is clear that the software was developed to permit this kind of analysis, but the public releases were not shipped with them. Thus, Ghidra's existing automated analysis is without regard to state and unconstrained, leaving researchers to mostly be restricted to manually inspecting the program after analysis to determine the behaviour themselves, when, in reality, many of the basic features researchers look for could be automatically found most of the time.

Schadenfreude was developed with one intention: to assist researchers in their analysis of complex software. As the original iteration [2] was developed with vulnerability researchers as the primary audience and userbase, the name *Schadenfreude* was chosen; the discovery of a vulnerability present in a software declared to be "secure" evokes elation from the achievement, the completion of ironic justice, and the joy in the misfortune of those believing it to be safe. To match the name, this iteration focuses less on specific features and instead provides an API through which to automate more of the trivial program analysis, allowing vulnerability researchers and reverse engineers to automate their common tasks and reduce the complexity of their more advanced research. This manifested with the use of the Z3 theorem prover and some basic program analysis to convert functions present in executables into a collection of assertions which modeled both data and control flow.

A. Contributions of Schadenfreude

There are two main contributions in this iteration of Schadenfreude: constrained bitvector representations of varnodes[1] and control flow modeling with Z3. Constrained bitvectors are fixed-length bit representations of values which may not necessarily have known concrete values such that individual bits or ranges of bits are restricted to specific values or by predicates which must evaluate to true for valid values for those bit ranges. These values are implemented using Z3's theory of bitvectors and models for Pcode [3] operations were defined as Z3 bitvector operations.

Control flow modeling is the process by which the control flow of a program is translated from instructions, to blocks of instructions and under what conditions one block of instructions leads to another (the "flow"), and finally into predicates in first-order logic which can be fed to Z3 for solving whether blocks are reachable and how it impacts and is impacted by the state of the program's data. As part of the modeling process, the conditions under which each block leads to another and thereby how the bitvectors representing data are constrained must be discovered in a process known as constraint discovery.

[1]The unit of data in Ghidra Pcode [3]

This process is crucial as, without it, control flow would be effectively unconstrained as there would be no indication of what cases would lead to a specific block of code being reached.

As an example of constraint discovery, consider the following function f:

```
f(a):
  // position 0
  if a == 0:
    c = 6 // position 1
  else if a % 3 == 0:
    c = 15 // position 2
  else:
    c = 9 // position 3
  return c // position 4
```

Fig. 1. An example program

In positions 0 and 4, a is unconstrained; there is no inherent property of a which must be true for the program to have reached either of those states assuming valid control flow was followed. On the other hand, at position 1, a is strictly equal to zero; at position 2, a is strictly non-zero and $a \equiv 0 \pmod 3$; at position 3, a is non-zero and $a \not\equiv 0 \pmod 3$. Control flow, in this way, can be used to discover constraints on varnodes and thus improve the effectiveness of stateful program analysis. The results of constraint discovery stage will be associated to varnodes such that they can be used by researchers and analysis softwares alike to have control flow relevant information on data in various contexts.

These contributions are critical as manual reviewers can have more context for a specific region and SMT solvers (e.g. Z3 [4], the solver used by Schadenfreude) or other advanced program analysis softwares can be used by researchers to develop analyses specific to their targets with significantly more capabilities than what are provided by Ghidra alone.

B. Contributions as a Result of Schadenfreude's Development

1) Z3 Contributions: Schadenfreude's development led to the contribution of nearly three thousand lines of source code to Z3[2]. These changes introduced a Java generic-based typesystem that both clarified and enforced datatype safety for statements in Z3 generated as a result of API calls both for types that are predefined and those defined at runtime. This was a critical improvement for Schadenfreude (to keep type consistency while generating data flow graphs) but also served as a general improvement for all uses of Z3 with minimal breaking changes.

2) Ghidra Contributions: The Sleigh language[3] definition for RISCV had inconsistencies in Pcode operation definitions that led to several issues during Schadenfreude testing. While not yet published (as these changes are still being verified), these changes improve the effectiveness of Ghidra when analysing targets compiled for RISCV.

[2]https://github.com/Z3Prover/z3/pull/4832

[3]https://ghidra.re/courses/languages/html/sleigh.html

C. Improvements Subsequent to Initial Presentation

At the time of the initial presentation, non-trivial acyclic control flow in programs was not possible to evaluate due to faulty modeling of control flow in Z3. The implications discussed in III-B were implemented to account for this, although Schadenfreude is still restricted to acyclic control flow graphs. Attempts were made to overcome the acyclic restriction, but this was simply not possible to accomplish in a week and is provably not solvable in the general case.

A representation for memory was implemented which allows Z3 to perform basic logic on stack operations, though heap is still not supported; this allowed for meaningful operations on buffers, although this is not demonstrated as it is still very unstable.

Z3 was also improved to be more ergonomic and precise, as mentioned in I-B1, which allows users of Schadenfreude to use the models of programs more effectively.

II. Related Work

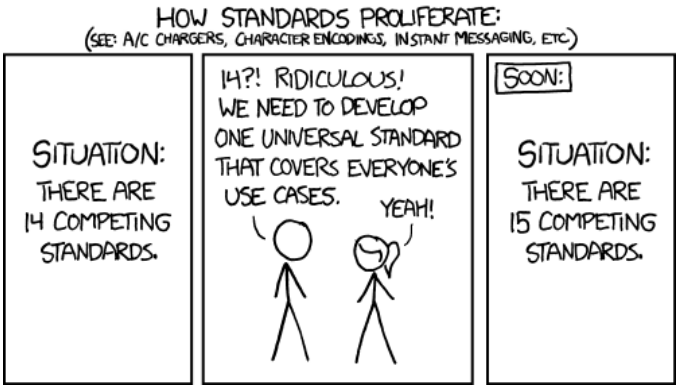

Fig. 2. Precisely what we wish to avoid. Source: [5]

A. angr [6] and related

angr, describing itself as a "both static and dynamic symbolic ("concolic") analysis [engine]," is the closest approximation to what this iteration Schadenfreude attempts to enable. angr uses the Keystone engine [7] for (dis)assembly, Claripy [8] for solving, and VEX IR [9] for intermediary representation; as requirements fulfilled by Keystone and VEX IR for angr are fulfilled by Ghidra and this iteration of Schadenfreude does not intend to handle solving, what remains is the constraint discovery and the implementation of SimEngine [10] to evaluate over the resulting constrained bitvectors.

As outlined in [6], Schadenfreude's approach will largely be based on graph-based vulnerability discovery and value-set analysis (though, instead of value sets, Schadenfreude uses constraint-set analysis with its bitvectors).

B. Ponce [11]

While angr attempts to be as general as possible in its implementation, Ponce is dedicated to simplifying symbolic execution and taint analysis specifically for reverse engineers. Developed as a plugin for IDA Pro, this software is limited to those who subscribe to IDA Pro and is thus not accessible to those who are not sponsored, able, or willing to pay for a subscription to IDA Pro [12]. Additionally, Ponce is limited to x86 (32- and 64-bit) and is mostly helpful for manual review. Unlike angr, it is quite ergonomic and simple analyses with Schadenfreude should be similarly simple to develop or use.

III. Method

A. Data Modeling

1) Constrained Bitvectors: As discussed in the introduction, the basic unit for all improvements will be the constrained bitvector. Similar to the three-value bitvector posed by existing program analysis plugins for Ghidra [13], these bitvectors represent both concrete and unknown states. Unlike basic three-value bitvectors, constrained bitvectors have no concrete representation and instead are represented by a set of assertions about ranges of bits.

The initial attempt to implement this datatype at the bit-level and attach it to existing varnodes was not successful, so instead custom Pcode operations were defined which assigned constraints to varnodes wherever control flow entered a region where a varnode was constrained. The results of this iteration were studied, then necessary changes were made such that constrained bitvectors were represented using Z3's theory of bitvectors instead of a custom implementation, which, as a side-effect, removed the need for custom Pcode operations. This was quite convenient as Ghidra has no mechanism by which to make inserted Pcode operations persistent[4].

Now that bitvectors were represented by Z3's theory of bitvectors, keeping track of constraints on bitvectors defined by uses of a constrained bitvector was no longer necessary to implement within Schadenfreude. This improvement positively impacted soundness as it no longer relied on a custom implementation and instead on a well-vetted implementation of the theory of bitvectors. Additionally, this trivialised symbolic memory.

2) Symbolic Memory: Now that bitvectors were represented by Z3's theory of bitvectors, memory could be represented as a two-dimensional Z3 array[5] which is implicitly sparse and indexed exactly as documented by Pcode's LOAD operation[6]. This memory could be mutated by passing the previous instruction's memory to the current, making memory state dependent on control flow. Unfortunately, this flow-based definition means that symbolic memory may be inconsistent in multithreaded applications as traditional data flow may not fully represent this [14]. Multithreaded applications will simply remain out of scope at this time as a solution for this single problem may be more difficult to implement and discover than the entireity of the changes proposed for this iteration of Schadenfreude.

Additionally, compared to a previous iteration, the memory representation no longer need to be endianness-aware. While relatively undocumented, some investigation showed that Ghidra implemented LOAD and STORE operations *strictly* using word-sized bitvectors. To implement this change, all that was necessary was to use the Ghidra API to discover the word size of memory.

3) Constraint Discovery: Helpfully, Ghidra provides a majority of the infrastructure to discover constraints via the CBRANCH (conditional branch) Pcode operation[7] wherein a boolean varnode is provided to input1. In this case, we can simply resolve the Pcode operations which led to the value of input1 and thus resolve the predicate itself that actually decided the result of the conditional branch. Thus, our constraints can be represented by assertions on the value present in input1.

Pcode which is provided as the result of decompilation in Ghidra has their own form of representing a varnode which has various different sources according to conditions - but does not include the conditions themselves (the MULTIEQUAL Pcode operation, which is equivalent to a phi node in single static assignment theory[8]). Each of the inputs provided to the MULTIEQUAL operation can be sourced to a specific block, and, due to an undocumented quirk of the Ghidra decompiler, inputs are in order of most to least constrained (i.e. in order of the number of branches followed in order to define the input)[9]. Using the rules for control flow defined in III-B, the output of a MULTIEQUAL operation can be defined by a chain of if-then-else statements whose conditions are based on whether the block which defines the input was hit. For example, with the predicate function `hit` which accepts a block number as an argument and returns a boolean indicating whether or not the block was hit, the return value of the program in figure 1 can be expressed as:

```
(ite hit(2) 15 (ite hit(3) 9 6))
```

B. Execution Modeling

In order to inform Z3 how to evaluate control flow and thereby data flow, execution must be modeled as a collection of implications on data based on whether blocks are hit. For example, consider the program in figure 1; operations are representable as implications based on whether or not a block was hit, resulting in the following set of implications:

```
(=> hit(1) (= c_1 6))
(=> hit(2) (= c_2 15))
(=> hit(3) (= c_3 9))
(=> hit(4) (= ret (ite hit(2) c_2
      (ite hit(3) c_3 c_1))))
```

[4]https://github.com/NationalSecurityAgency/ghidra/issues/401

[5]https://rise4fun.com/z3/tutorialcontent/guide#h26

[6]https://ghidra.re/courses/languages/html/pcodedescription.html#cpui_load

[7]https://ghidra.re/courses/languages/html/pcodedescription.html#cpui_cbranch

[8]https://ghidra.re/courses/languages/html/additionalpcode.html

[9]This is because MULTIEQUAL inputs are discovered by BFS, which is coincedentally exactly the method needed to discover most branches followed.

With operations now represented as the result of implications of blocks being hit, control flow can be represented as the implications of edges being followed on blocks being hit[10]. To accomplish this, the following rules were derived:

1) A given edge being followed implies both the source block and the destination block are hit.
2) A given block being hit implies exactly one of the input edges were followed, if it has any.[11]
3) A given block being hit implies exactly one of the output edges are followed, if it has any.
4) A given edge being followed implies the constraint on that branch evaluates to true.

Using these rules, f as defined in figure 1 can now be expressed as the following collection of assertions[12]:

```
; operations
(=> hit(1) (= c_1 6))
(=> hit(2) (= c_2 15))
(=> hit(3) (= c_3 9))
(=> hit(4)
  (= ret (ite hit(2) c_2
      (ite hit(3) c_3
          c_1))))

; control flow
(=> hit(0) (one-of edge(0 1)
                   edge(0 2)
                   edge(0 3)))
(=> edge(0 1) hit(0))
(=> edge(0 1) hit(1))
; ... more of rule 1
(=> edge(0 1) (= a 0))
(=> edge(0 2) (and (= (mod a 3) 0)
                   (not (= a 0))))
(=> edge(0 3) (not (or (= (mod a 3) 0)
                       (= a 0))))

(=> hit(1) (one-of edge(0 1)))
(=> hit(1) (one-of edge(1 4)))
(=> edge(1 4) hit(1))
(=> edge(1 4) hit(4))
; ... same for 2, 3

(=> hit(4) (one-of edge(1 4)
                   edge(2 4)
                   edge(3 4)))
```

[10]By stating everything in terms of implications based on uninterpreted predicates, it is possible to set Z3 to use Horn logic and prove some inductive facts, though this feature was not used for this iteration of Schadenfreude. For more information, see https://rise4fun.com/Z3/tutorial/fixedpoints. This mechanism will likely be used to unroll cycles in the future.

[11]This rule and the next restricts the current version of Schadenfreude to control flow graphs which are strictly directed acyclic, but this can (and will) be changed in the future.

[12]This is informally expressed; the actual implementation uses quantifiers to perform the `one-of` operation and constant functions to represent whether blocks/edges are hit/followed. This representation should be interpreted as descriptive, not prescriptive.

Then, to evaluate, `hit(0)` and `hit(4)` are asserted as true. An assertion to the value of `a` can be made, and, simply by acquiring the model resolved by Z3, one can evaluate `ret`. Moreover, one can prove the functional equivalency of this function to another by simply asserting that inputs and return values of this function are equivalent to another[13].

IV. Demonstration

A demonstration has been prepared for reproducing the claims made in III which can be viewed by executing `demo.sh` provided in the `schadenfreude.tar.gz` sources archive.

In order to demonstrate the soundness of Schadenfreude across multiple architectures, the demonstration will identify and use all compilers on the system. To try Schadenfreude with another architecture, simply install its cross-compiler. Schadenfreude has primarily been tested with x86_64-linux-gnu-gcc, aarch64-linux-gnu-gcc, and riscv64-linux-gnu-gcc (see I-B2 for details on RISCV programs), though should be theoretically compatible with any architecture which has a correct Sleigh language implementation for Ghidra.

This demonstration implements the following use cases of Schadenfreude:

1) Evaluation of functions present in an executable for arbitrary valid inputs
2) Proving statements about data flow based on control flow
3) Disproving statements about data flow based on control flow
4) Proving statements about control flow based on data flow
5) Functional equivalency proofs

The demonstration uses the following C source, compiled as part of a shared object library:

```
#include <stdio.h>

int diamond(int x, int y) {
    int c = x + y;
    if (c == 0) {
        return c;
    } else if (x == 5) {
        c += 15;
    } else if (y == 5) {
        c += 32;
        puts("y == 5");
    } else {
        c += 6;
    }
    return c;
}
```

For more information about how this demonstration implements each of the use cases listed, please run the demonstration or view `demo.log`, provided in the `schadenfreude.tar.gz` sources archive.

[13]While not explicitly discussed here, an instance of this can be found in the demonstration code.

V. Final Thoughts

While Schadenfreude: Resurrection is not quite in the state that I had hoped it would be at this point, I have a better respect for the complexity of using SMT solvers in program analysis and believe the submission provided is demonstrably quite powerful. In its current state, it is extremely effective against functions whose control flow graphs are acyclic. This is, unfortunately, very few programs.

I developed this for my own projects, and will continue to develop it further. My likely next step is control flow graph cycle unrolling, which frequently leads to path explosion. Then, in the next iteration, implementing some basic archetypes for cycles (for with bounds, while, do-while, etc.), and when that inevitably encounters an issue, I'll look into a solution, so on and so forth. I recognise that, because of the halting problem, I'll never have a general solution – but that's not going to stop me from shaving down the unsound/incomplete cases!

References

[1] "Ghidra," https://ghidra-sre.org/, National Security Agency, Feb 2020.

[2] A. Crump and T. Heinen, "Schadenfreude," Aug 2020, Available upon request.

[3] "P-Code Reference Manual," https://ghidra.re/courses/languages/html/pcoderef.html, National Security Agency, Feb 2020.

[4] "The Z3 Theorem Prover," https://github.com/Z3Prover/z3, Microsoft Research, Oct 2020.

[5] "Standards," https://xkcd.com/927/, xkcd, Jul 2011.

[6] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[7] N. A. Quynh, "KEYSTONE: Next Generation Assembler Framework," https://www.keystone-engine.org/docs/BHUSA2016-keystone.pdf, Aug 2016.

[8] "Claripy," https://docs.angr.io/advanced-topics/claripy, angr, 2019.

[9] "Intermediate representation," https://docs.angr.io/advanced-topics/ir, angr, 2018.

[10] "Execution engine," https://docs.angr.io/core-concepts/simulation, angr, 2020.

[11] A. G. Illera and F. Oca, "Ponce," https://github.com/illera88/Ponce, 2016.

[12] "Support for IDA Free 7," https://github.com/illera88/Ponce/issues/103, Jan 2020.

[13] "GhidraPAL/TVLBitVector.java," https://github.com/RolfRolles/GhidraPAL/blob/master/src/main/java/ghidra/pal/absint/tvl/TVLBitVector.java, Aug 2019.

[14] J. C. Brodman, B. B. Fraguela, M. J. Garzarán, and D. Padua, "New abstractions for data parallel programming," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'09. USA: USENIX Association, 2009, p. 16.