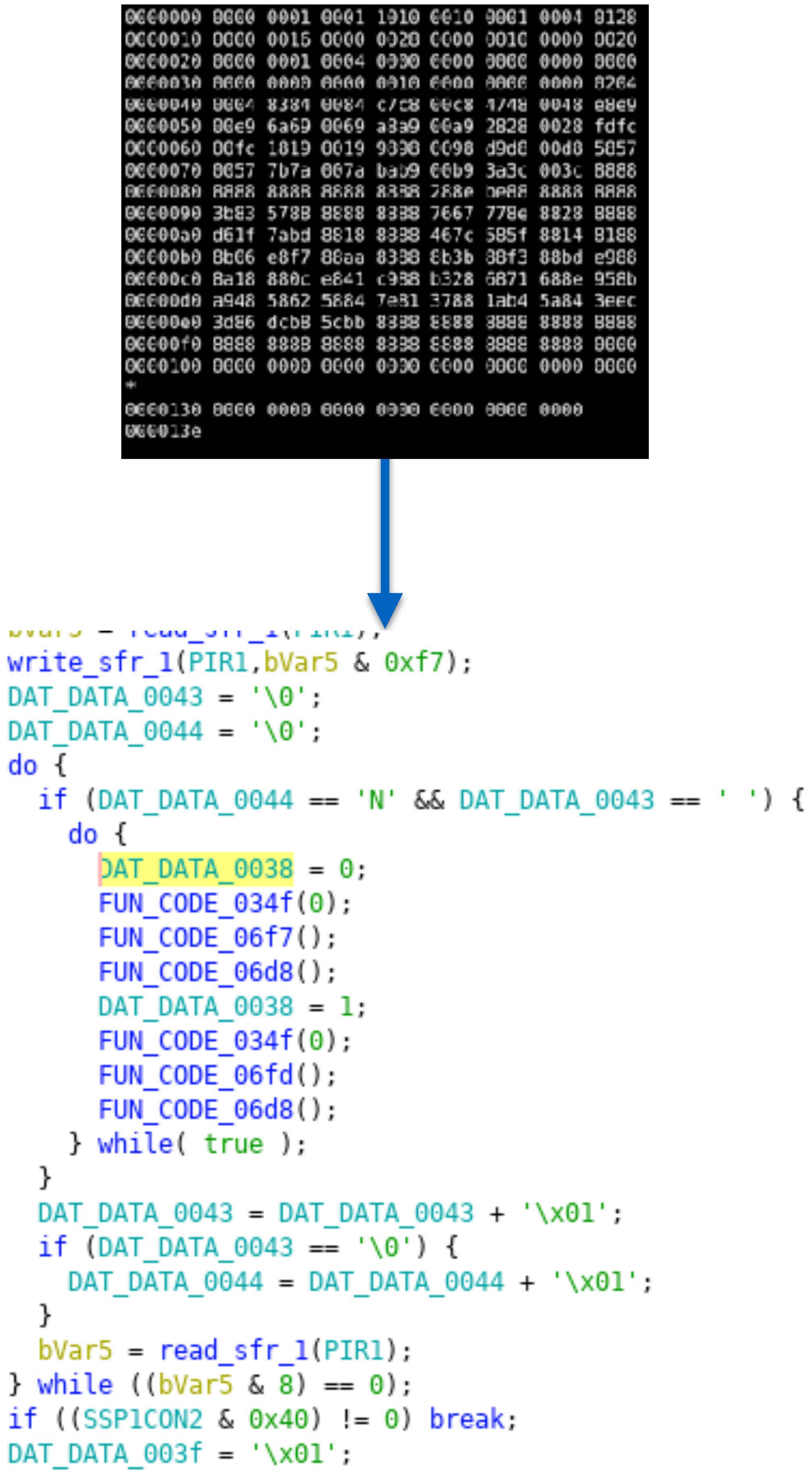
Scaling Binary Analysis

Kris Micinski (Syracuse)
Yihao Sun, Chang Liu, and collaborators

General stage for this work…

- Binary analysis: static/dynamic analysis of arbitrary binary code
- Reverse engineering: recover readable source from binary
- Decompilation: automatically derive source from binary



```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0020 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a3a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9398 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8388 788e 0e88 8888 8888
0000090 3b83 5788 8888 8388 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8388 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8388 8888 8888 8888 8888
00000f0 8888 8888 8888 8388 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

```
write_sfr_1(PIR1,bVar5 & 0xf7);
DAT_DATA_0043 = '\0';
DAT_DATA_0044 = '\0';
do {
  if (DAT_DATA_0044 == 'N' && DAT_DATA_0043 == ' ') {
    do {
      DAT_DATA_0038 = 0;
      FUN_CODE_034f(0);
      FUN_CODE_06f7();
      FUN_CODE_06d8();
      DAT_DATA_0038 = 1;
      FUN_CODE_034f(0);
      FUN_CODE_06fd();
      FUN_CODE_06d8();
    } while( true );
  }
  DAT_DATA_0043 = DAT_DATA_0043 + '\x01';
  if (DAT_DATA_0043 == '\0') {
    DAT_DATA_0044 = DAT_DATA_0044 + '\x01';
  }
  bVar5 = read_sfr_1(PIR1);
} while ((bVar5 & 8) == 0);
if ((SSP1CON2 & 0x40) != 0) break;
DAT_DATA_003f = '\x01';
```

- Serious effort expended to scale program analyses to huge code in high-level PLs (e.g., DOOP, [Smaradgakis et al.])
- Big lie in security: lots of work meant to be applied to binaries *requires source in practice*
  - Fuzzing, symbolic execution, flow analyses, …
- Binary analysis *tools* not informed by state-of-the-art in program analysis construction methodology
  - Decompilers are ad-hoc, heuristic, and can be brittle in practice—especially for maliciously-construed code

Today's Talk

- A little bit about some mostly-completed systems work
- An introduction to modern disassembly and binary analysis
- New directions in binary rewriting based on state-of-the-art analysis techniques

- **Assemblage**—binary corpus construction system
    - How do we reliably, reproducibly, and scalably build diverse binary corpuses?
- **Reversi**—Declarative binary instrumentation
    - Binary instrumentation is ad-hoc, expensive, and brittle
    - Many binary analysis tools are collections of heuristics intended to implement a static analysis of a binary
    - Common structure: lift binary up to some level (functions, structs, …), observe structure (RE) / perform transformation

Assemblage, in a slide

- An extensible, automated **binary corpus generation** tool
- Discovers, configures, and builds source software repositories (GitHub) in wide variety of configurations
- Scales across nodes of different configurations (Linux + Windows + …); work distribution via message queue
- Post-processing binary to mine metadata (e.g., source-to-binary symbol mapping) for applications in ML and beyond
- Currently running on 2 Linux worker (20cores) and 10 Windows (4cores) worker VMs, thousands of binaries / day

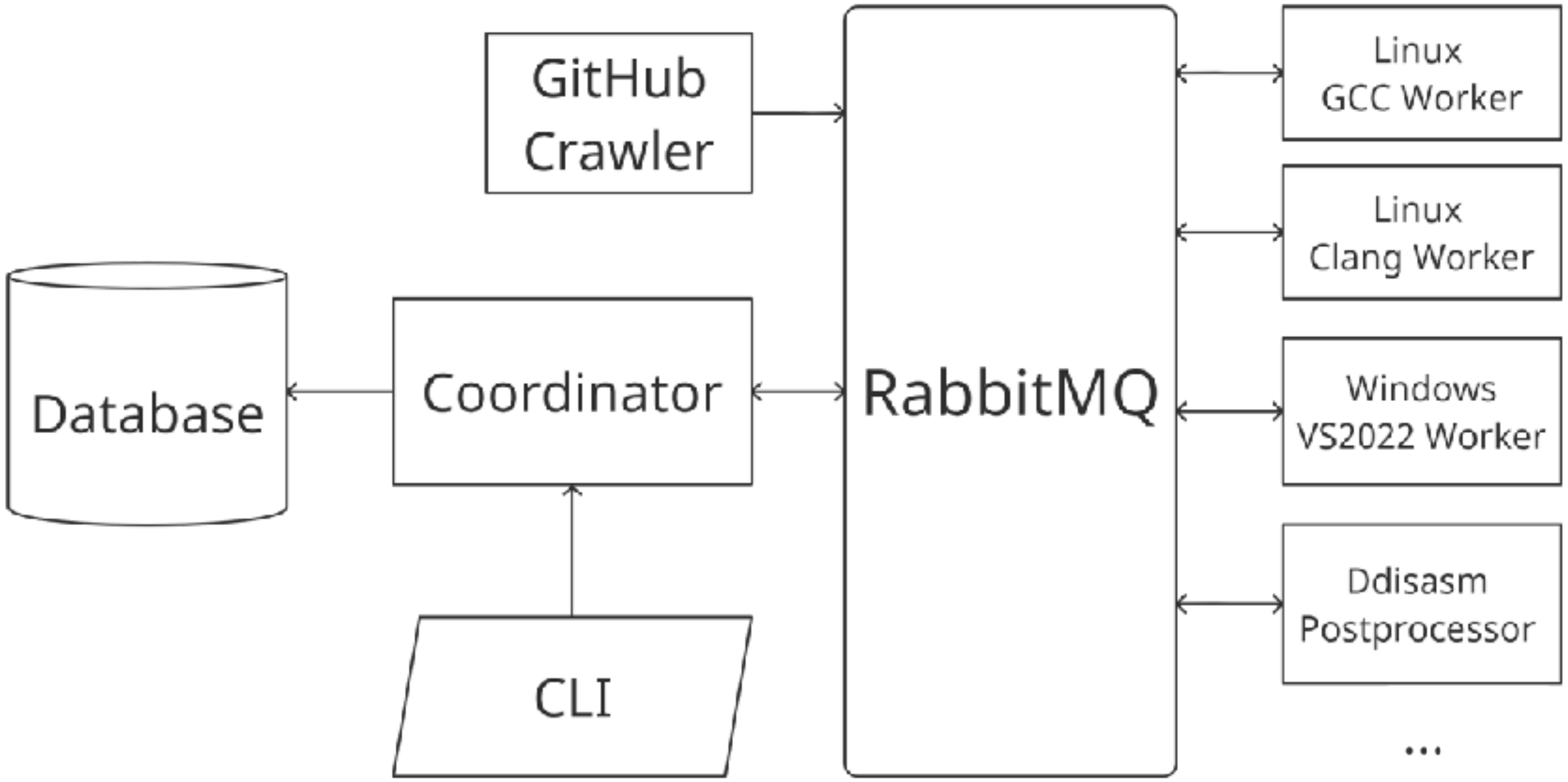
Motivation — Curated Binary Corpuses

- **Application** — Want to **train good models** to achieve binary analysis / classification tasks:
  - Malware classification, precise-but-fast disassembly, variable name synthesis (RE), entrypoint identification,..
- **Challenge**—getting diverse input binaries; avoid, e.g., overfitting due to specific compiler choices
- **Solution**—automated tool to build binary corpus from source repos in a variety of build configurations

High-Level Principles

- (Safety) Run builds inside Docker/VM
- (Distribution) Centralized *coordinator* node, multiple *workers* all run common worker task
- (Persistence) MySQL database syncs system state (repos discovered, built, etc…), binaries warehoused via NFS
- (Scalability) Message queue facilitates communication between coordinator to worker
- (Configurability) Implementation provides many distinct build configurations (Linux/Windows/x86-32bit/amd64/…), extensible framework

Architectural Diagram



Coordinator

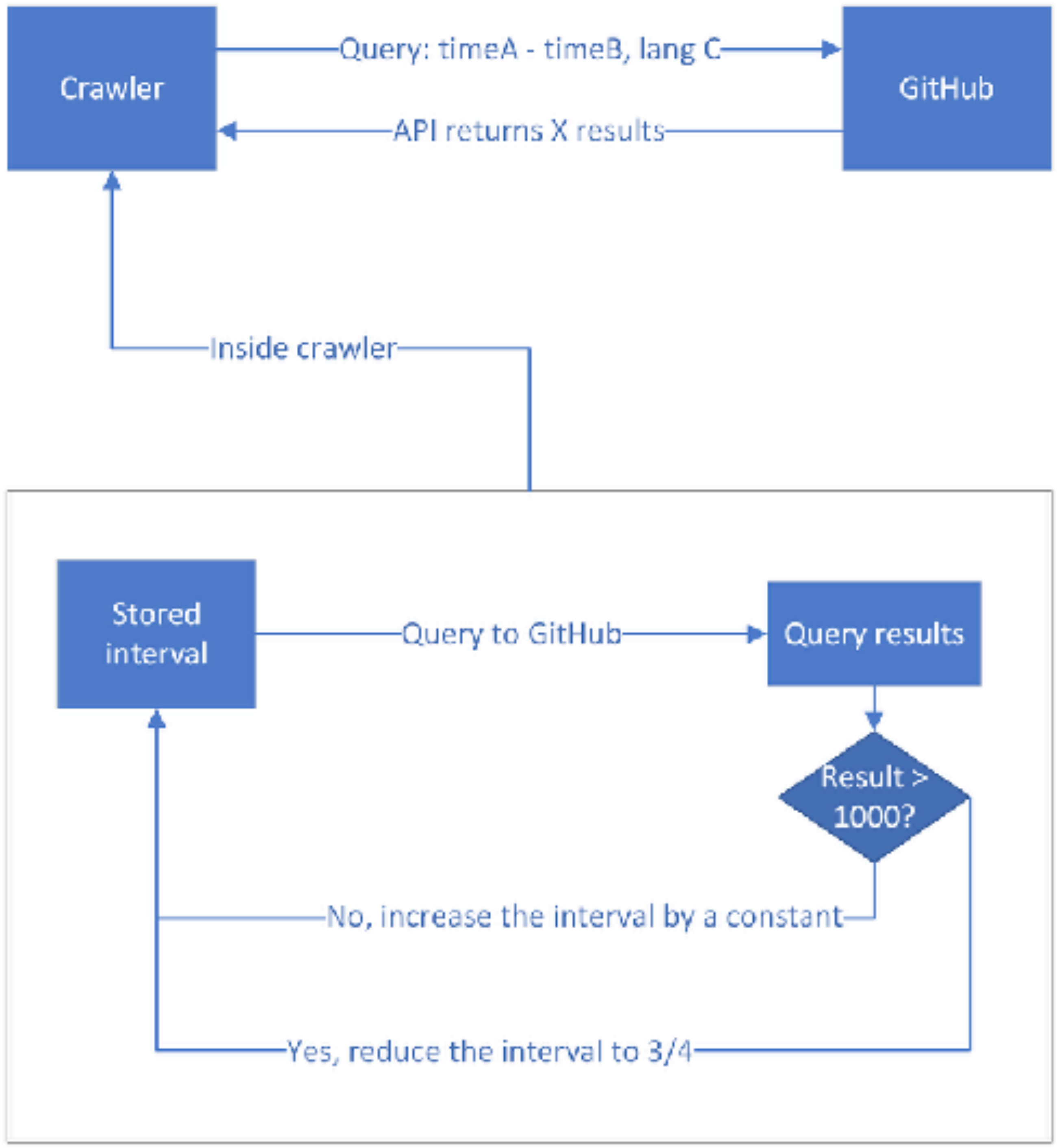
- Controls system coordination—one to many architecture
- Distributes tasks to workers, maintains overall system state in MySQL DB (SQLite was too slow for us)
- Is scalability a concern? We believe no (up to 10s of nodes), as many tasks involve high amounts of compute time relative to comm
    - E.g., building single repo takes ~30-120 seconds vs. 3-4 messages exchanged
    - We have seen RMQ scale to 100s of nodes previously

Workers

- Run on many nodes (possibly same as coordinator), fetch work from coordinator, send results to coordinator
- Diversification, cloning, building, and post-processing
- Implemented in Python, but not limited by Python's threads—multiple worker processes communicate via MQ
- Two worker *platforms* (Linux/Windows) share large amount of overlapping code (abstract classes)—adding new platforms (e.g., BSD) are possible without much work

Assemblage's Crawler — GitHub

- Query GitHub for repos with specific filters to retrieve 1k results
- Obtain diversity by specifying a precise timestamp—can get many repos by varying timestamp or its resolution
- Dynamically tune timestamp based on results (always attempt to get 1k repos)
- Respect rate limits



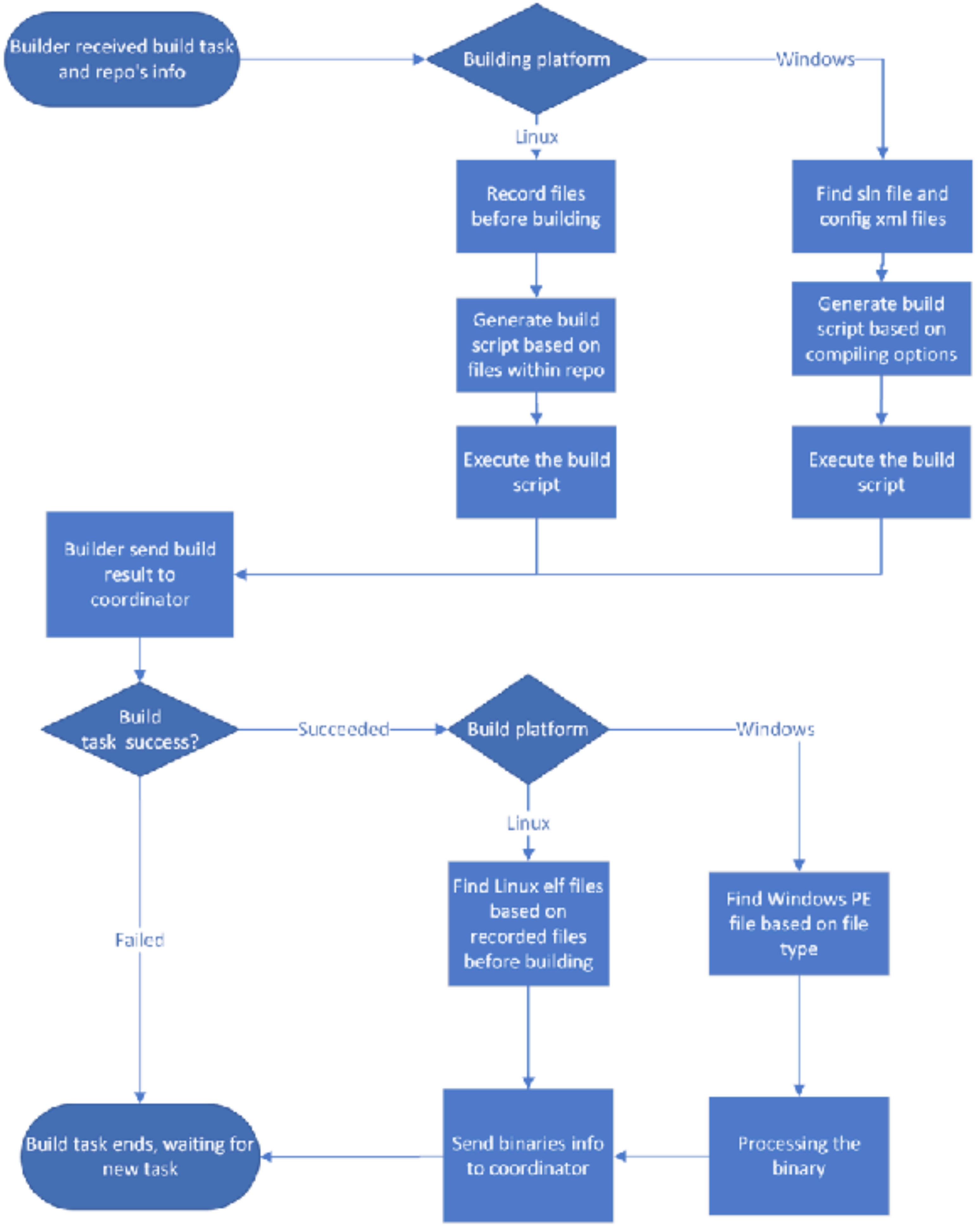
Crawling — Prior Work: GHTorrent

- Attempts to scalably mine GitHub to create a mirror
- Monitors public events, crawls dependencies
- We began with GHTorrent as “seed” database; have been implementing *our own* crawler useful for our needs
- We found GHTorrent ineffective at quickly finding many Windows repos (we want Windows PE binaries for C/C++)
- However, we continue to borrow ideas (e.g., exhaustive crawling) from GHTorrent

Build Task

- Implementations for several **build platforms**—currently Linux and Windows
- Within a build platform, can configure several build **variants**
    - E.g., Windows MSVC++ versions, x86/x86_64/…
- For Linux, we have several toolchains for each variant—make/llvm, cmake/llvm, etc…
    - Rough pass over repo to determine which variants make sense (e.g., can't use Make/cmake without Makefile/CMakeLists.txt)
- Most focused on Windows MSVS (no Borland, etc.. future step?)

Building Flowchart



Builder received build task and repo's info
Building platform
Windows
Linux
Record files before building
Generate build script based on files within repo
Execute the build script
Find sln file and config xml files
Generate build script based on compiling options
Execute the build script
Builder send build result to coordinator
Build task success?
Succeeded
Build platform
Windows
Linux
Find Linux elf files based on recorded files before building
Find Windows PE file based on file type
Failed
Build task ends, waiting for new task
Send binaries info to coordinator
Processing the binary

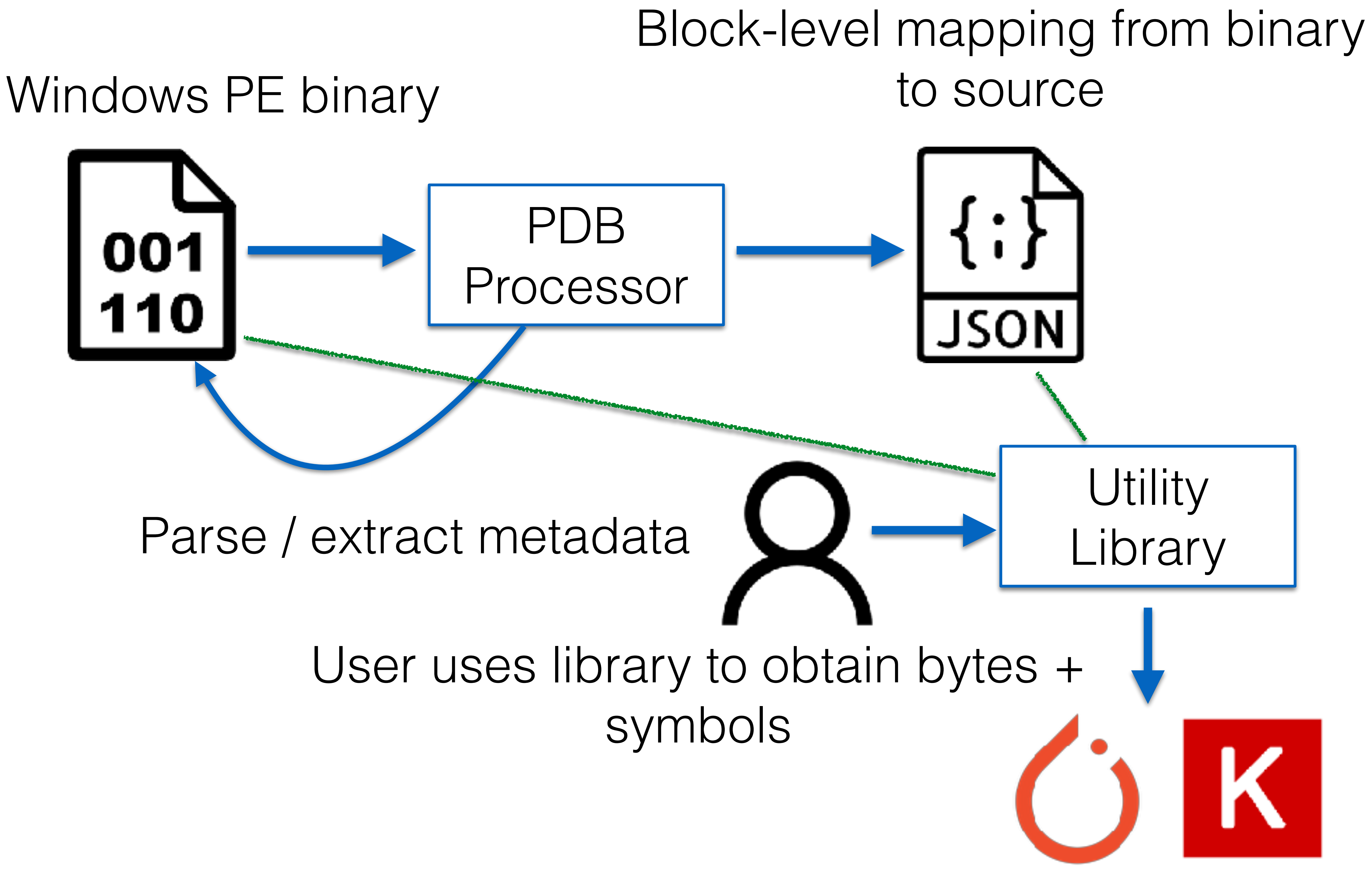
Build Failure/Success

- Build failures occur for a variety of reasons
    - Their fault—code incomplete, broken Makefile, …
    - Our fault—improper build system, don't have correct libraries/SDKs, …
- Assemblage infrastructure collects rough stats about repos to determine build systems
- **Our solution**—assume build images include lots of SDKs / packages; some things (e.g., those requiring specific library versions) may not work. We continue to monitor and refine

Postprocessing

- After building binaries, we index each to maintain a mapping from build configuration to binary
- For Windows binaries, special metadata collection phase
- Pluggable implementation allows multiple post-processing steps to be run
  - E.g., PDB parsing, ddisasm pass, …
- **(Future?)** Obfuscation passes, further binary diversification?

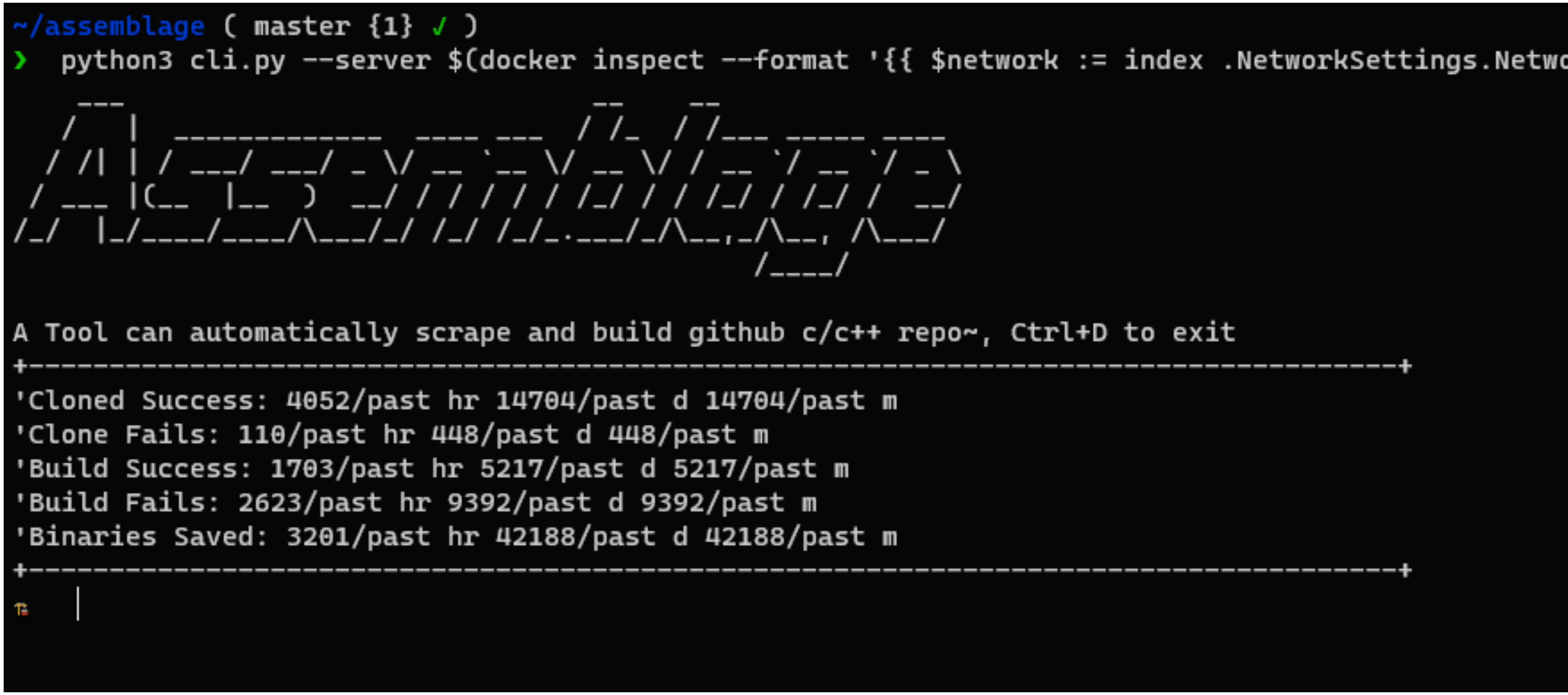
PDB-Based Source-to-Symbol mapping


Block-level mapping from binary to source
Windows PE binary
001
110
PDB Processor
JSON
Utility Library
Parse / extract metadata
User uses library to obtain bytes + symbols
K

```
{
    "Platform": "x86" or "x64",
    "Build_mode": "Debug" or "Release",
    "Source_url": Github repo link,
    "Toolset_version" : Visual Studio tool set version,
    "Optimization" : "Od/O1/O2",
    "Binary_info_list":
      [
         {
              "file": binary_file,
              "functions":[
                  {
                        "function_name": function_name,
                        "intersect_ratio": injected debug pin ratio,
                        "source_file": source_file_name,
                        "function_info":[
                            {
                               "rva_start": rva start address,
                               "rva_end": rva end address,
                               "debug_ratio" : injected debug pin ratio,
                            }
                        ],
                        "lines":[
                            {
                               "line_number" : line number,
                               "rva": RVA address,
                               "length": length of line in binary,
                               ("source_file": source file path)
                           }
                       ],
```

Command-Line Interface

- We include CLI for interfacing with Assemblage's coordinator—allows monitoring high-level system status and controlling coordinator



```
~/assemblage ( master {1} ✓ )
❯  python3 cli.py --server $(docker inspect --format '{{ $network := index .NetworkSettings.Netwo
    ___                               __    __
   /   |  _____________  ____ ___  / /_  / /___ _____ ____
  / /| | / ___/ ___/ _ \/ __ `__ \/ __ \/ / __ `/ __ `/ _ \
 / ___ |(__  |__  )  __/ / / / / / /_/ / / /_/ / /_/ /  __/
/_/  |_/____/____/\___/_/ /_/ /_/_.___/_/\__,_/\__, /\___/
                                              /____/

A Tool can automatically scrape and build github c/c++ repo~, Ctrl+D to exit
+--------------------------------------------------------------------------------+
'Cloned Success: 4052/past hr 14704/past d 14704/past m
'Clone Fails: 110/past hr 448/past d 448/past m
'Build Success: 1703/past hr 5217/past d 5217/past m
'Build Fails: 2623/past hr 9392/past d 9392/past m
'Binaries Saved: 3201/past hr 42188/past d 42188/past m
+--------------------------------------------------------------------------------+
```

Warehousing and Archiving

- Binaries dumped to large folder, organized by hash
- Directory of directories: first two letters of hash, followed by folder with rest of hash
- Each binary contains its build artifacts (object code + PDB if present) along with JSON file (Windows builds)
- Each binary proper archived as .tar.gz file
- Now moving to NFS-based warehousing (almost complete)

Results / Status

- Past year: building out core facilities for system, focus on scalability and build success. Specific focus on Windows

- **Deployed**—we run Assemblage 24/7 at SU on 10 (4-8 core, 8GB RAM) Windows nodes and 2 Linux nodes

- **Now**: building 30k binaries per day (Windows); tens of thousands of Linux binaries per day

- Crawling / cloning / building / dumping working, but we continue to fix bugs as we scale up

Reproducibility

- Tested reproducibility of Assemblage to rebuild a dataset of ~30k Windows binaries
- Diffed disassembly of all `.text` sections from binaries
- Vast majority (99%) have significant overlap (>99%); we believe others are due to compilation-time randomness of -O3 and etc…
    - But, still measuring this and brainstorming eval metrics
        - Information-theoretic distance metrics (LZJD), edit distance, …

Immediate Next Steps/Goals

- Next two months—scale Assemblage to generating 10-100k binaries per day (almost there, some further optimization, we are booting up more workers)
- Using current resulting binaries from Assemblage to train conventional models for malware classification
- “Open-source” release of Assemblage—we plan for Assemblage to be ready for collaborators to use in several weeks (though we’re happy to give away now)
  - Will likely save true open-source announcement until after paper submission
- VLDB DS paper submission

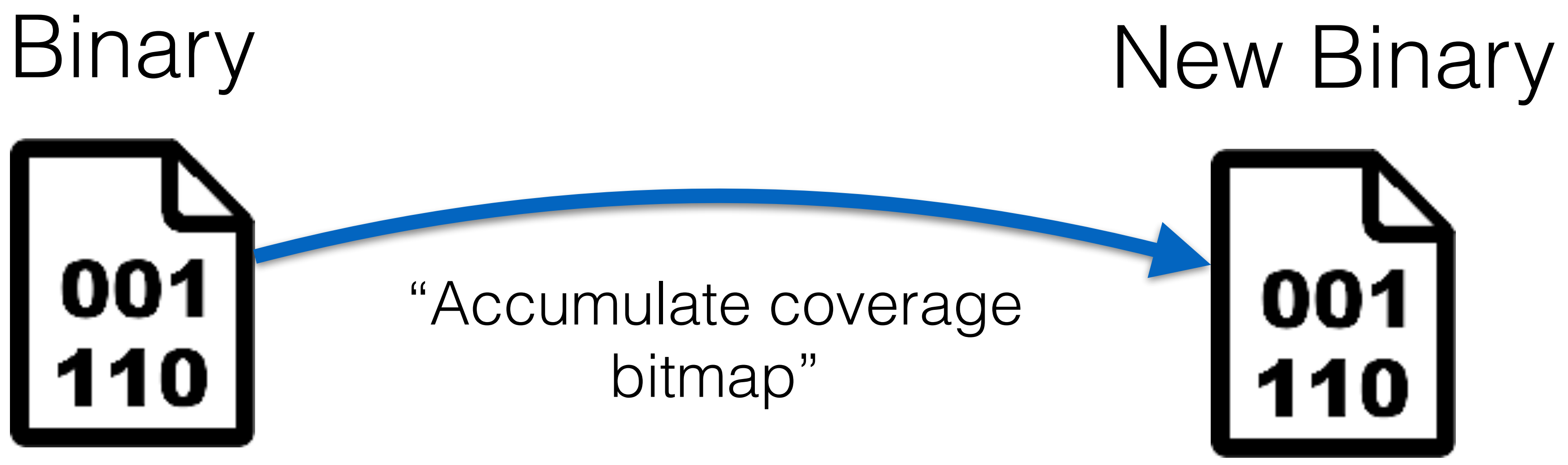
Medium-Term Next Steps

- Novel directions in binary analysis via ML, static / dynamic analyses, …
- Use Assemblage as baseline to obtain diversity
- Pluggable infrastructure in Assemblage allows rapidly generating new datasets by extending Assemblage's core infrastructure
    - Binary obfuscation (e.g., obfuscator-llvm), stripping symbol tables, …
- Even more flexibility in Assemblage (other repo sources, more configuration options, updating web UI)

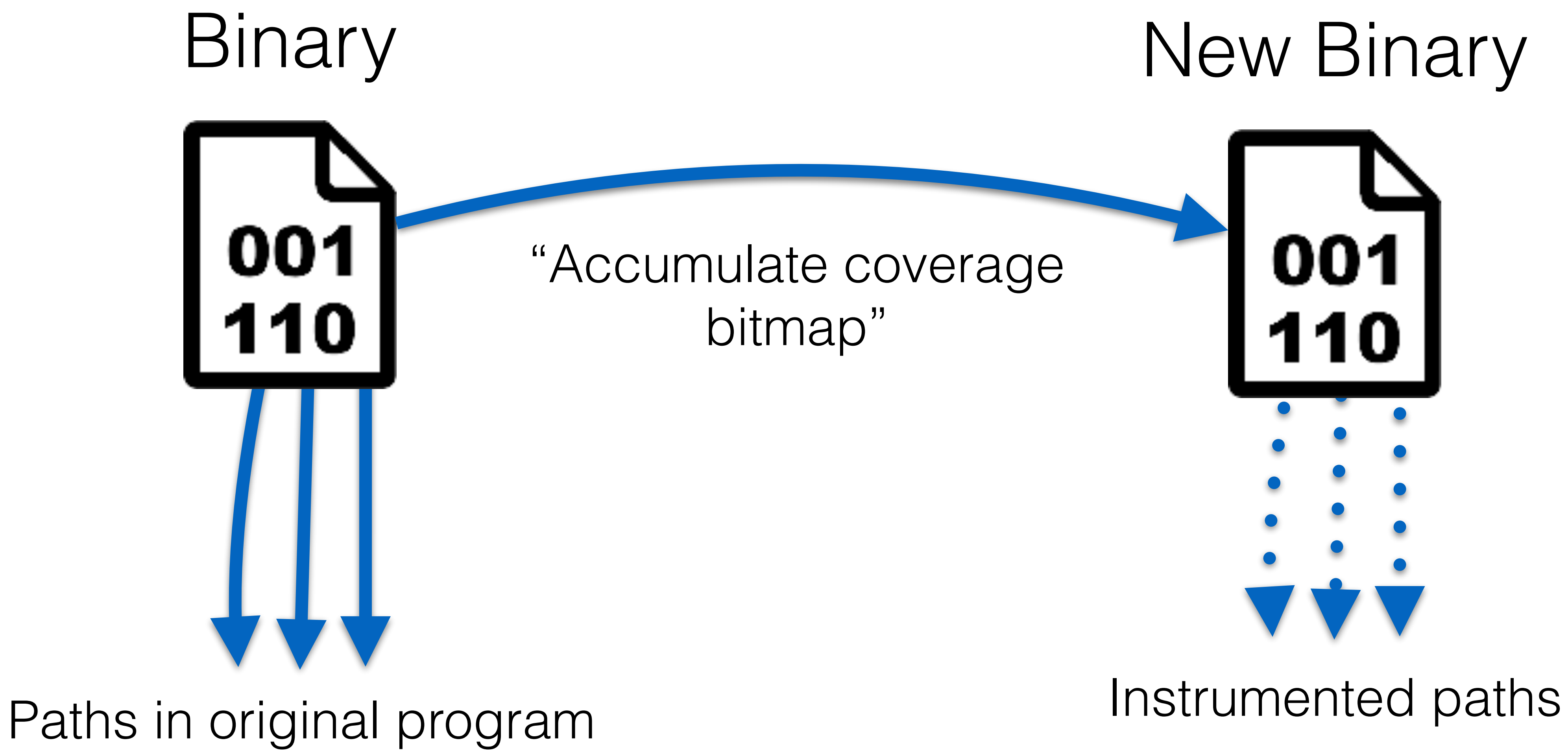
Assemblage Wrap-up

- Assemblage was built to compile large binary corpuses
- A specific focus is on build diversity with labelled metadata for the purposes of training ML-based binary analysis tools
- Extensible and variadic build system configures (and post-processes) GitHub repositories
- Designed from the ground up to scale to (at least) tens to 100s of cores, collecting ~10-100k binaries per day
- We are currently running Assemblage to generate ~10k binaries / day; hoping to push to 100k soon!

Binary Instrumentation



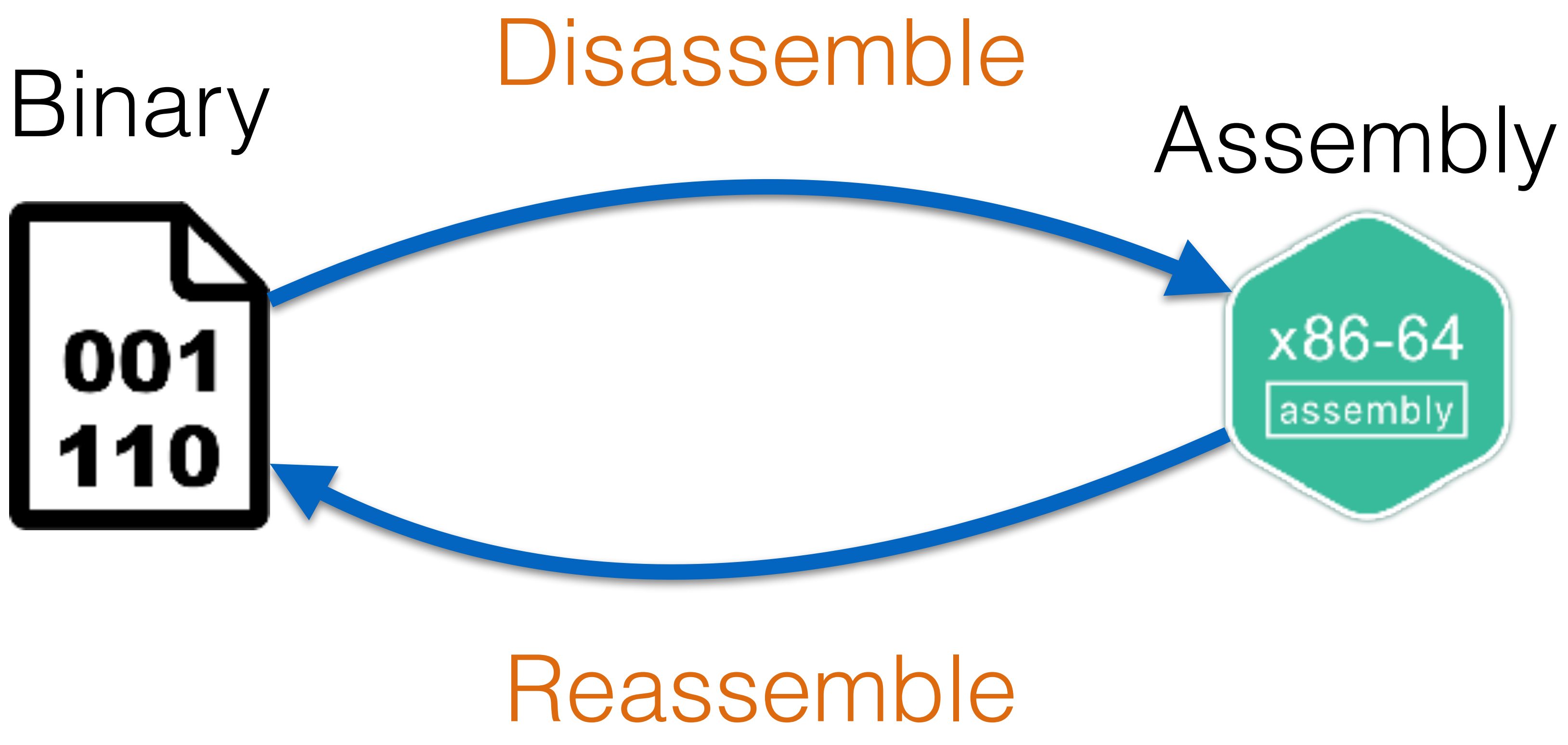
Binary Instrumentation



Intermezzo—Disassembly

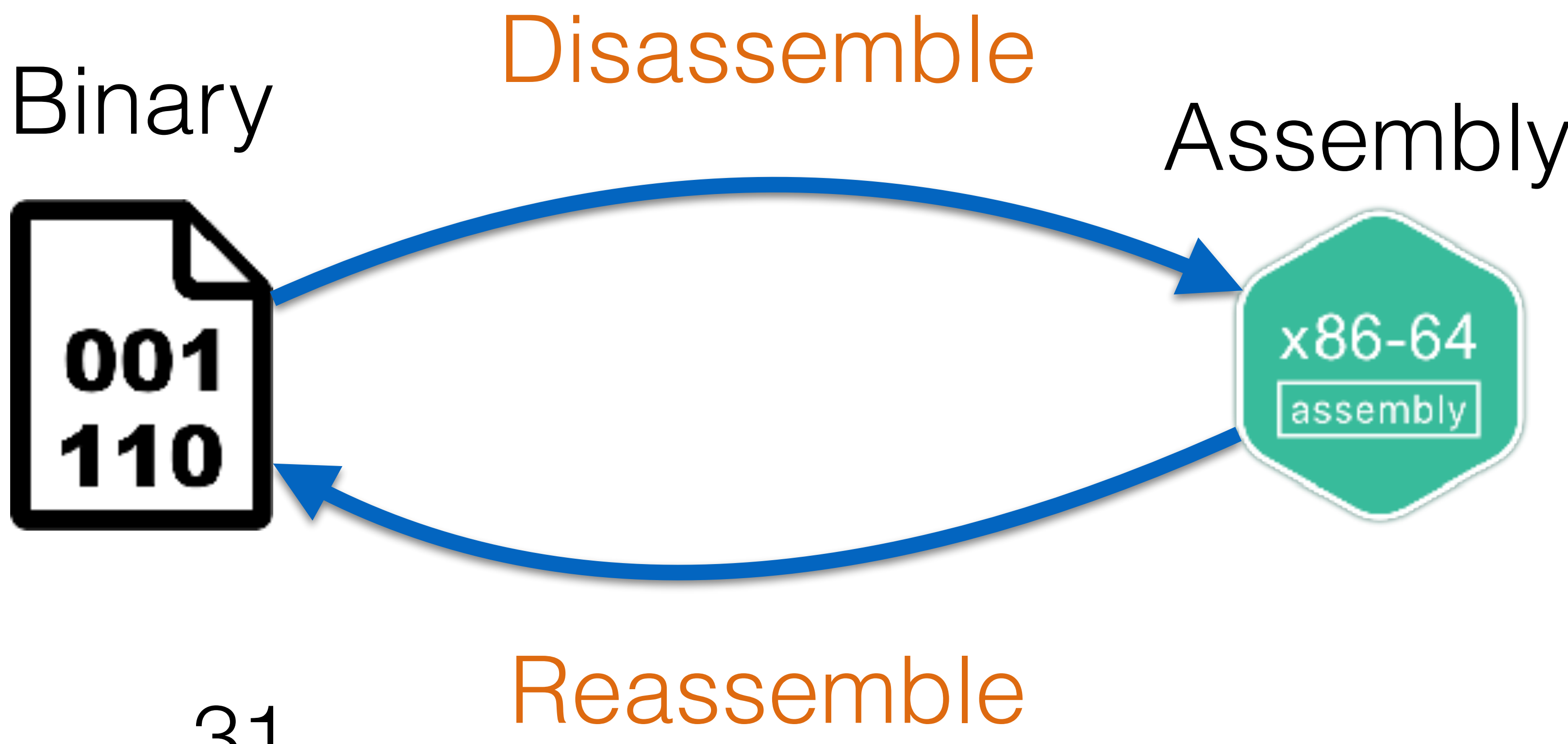
- Disassembly—here are bytes, give me structured assembly
    - Sections (text/data/bss/…), entrypoints (funcs), labels (BBs)
- Did you know: disassembly is **hard**
    - Stripped binaries; even identifying *entrypoints* challenging
- Disassembly *has* to be sound:
    - No high-level analysis works over a literal binary; in practice, many lift binary to asm/C to perform analysis
    - Unsoundness in disassembly = unsoundness in analysis!

Reassembleable assembly

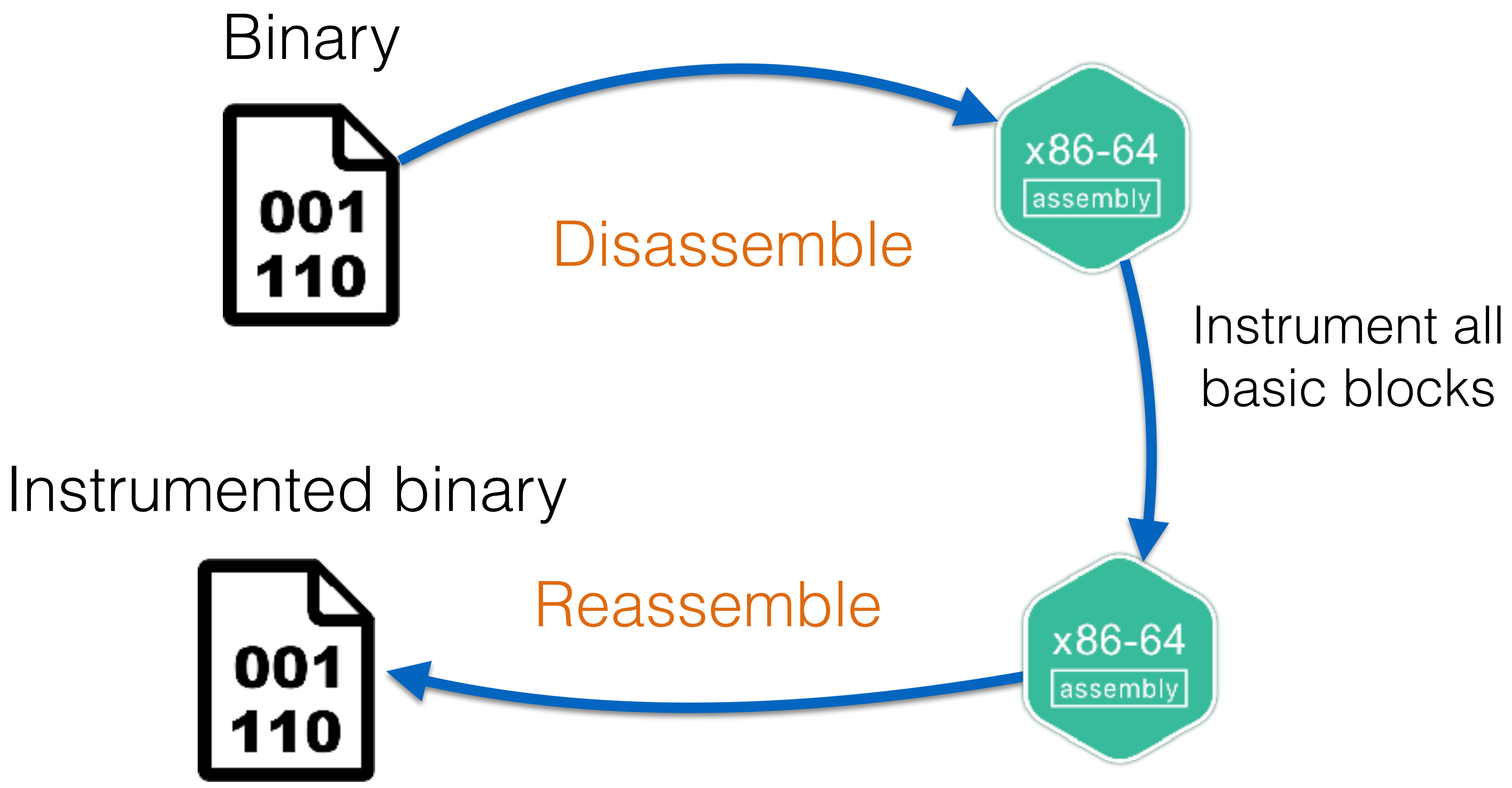


We want…
Reassemble ◦ Disassemble = Id

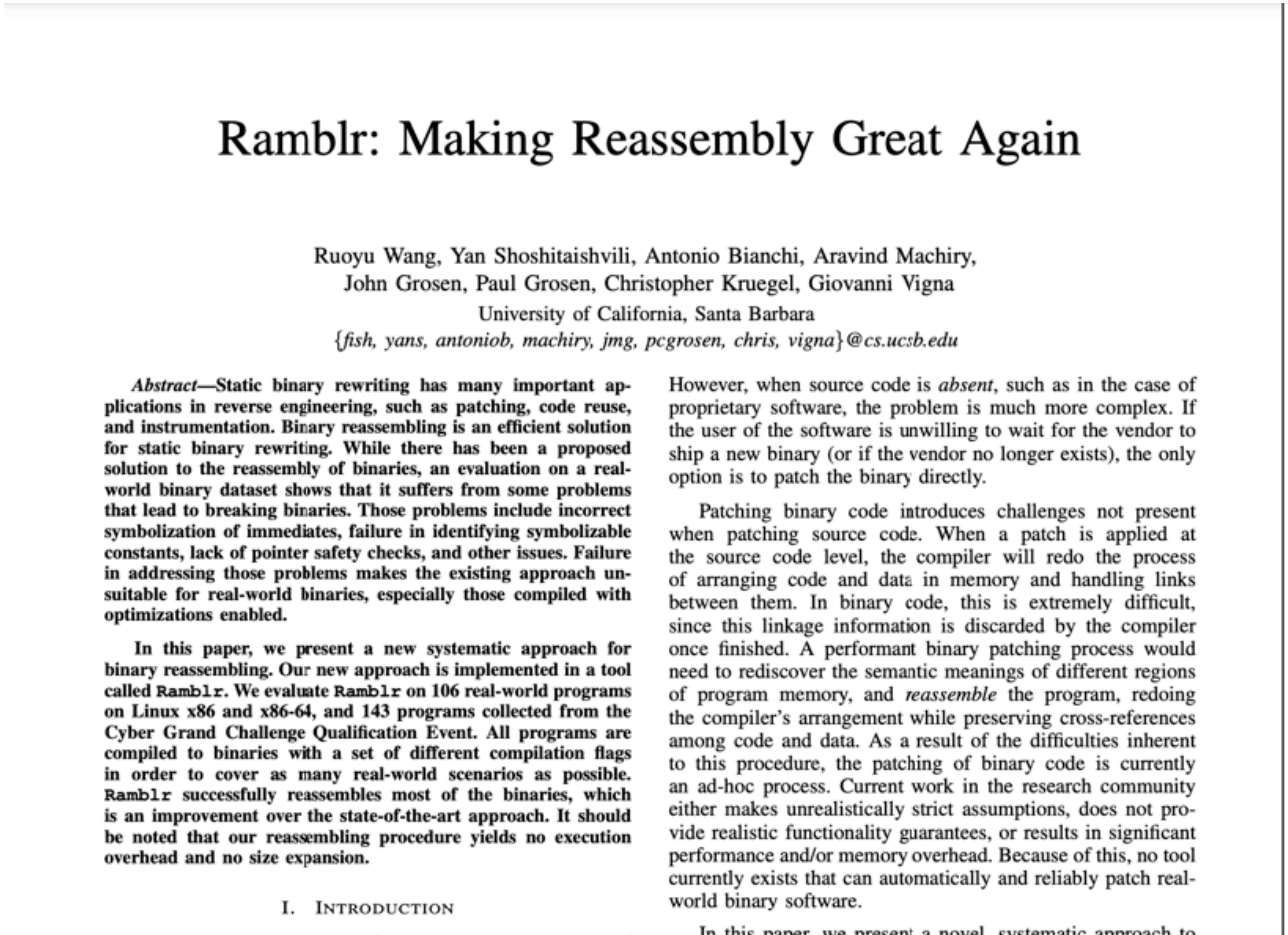
- I claim: reassembleable disassembly is a *prerequisite* for any sound binary analysis toolchain
- (At least those based on inferences based on the assembly, i.e., *all of the ones familiar to us*)
- But no production tools truly based upon reassembleable assembly (capstone, etc…)



- Also, this gives us a hook to begin applying **transformations** based on the assembly



~99% correctness for reassembleable disassembly



Ramblr: Making Reassembly Great Again

Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry,
John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna
University of California, Santa Barbara
{*fish, yans, antoniob, machiry, jmg, pcgrosen, chris, vigna*}*@cs.ucsb.edu*

***Abstract*—Static binary rewriting has many important applications in reverse engineering, such as patching, code reuse, and instrumentation. Binary reassembling is an efficient solution for static binary rewriting. While there has been a proposed solution to the reassembly of binaries, an evaluation on a real-world binary dataset shows that it suffers from some problems that lead to breaking binaries. Those problems include incorrect symbolization of immediates, failure in identifying symbolizable constants, lack of pointer safety checks, and other issues. Failure in addressing those problems makes the existing approach unsuitable for real-world binaries, especially those compiled with optimizations enabled.**

In this paper, we present a new systematic approach for binary reassembling. Our new approach is implemented in a tool called `Ramblr`. We evaluate `Ramblr` on 106 real-world programs on Linux x86 and x86-64, and 143 programs collected from the Cyber Grand Challenge Qualification Event. All programs are compiled to binaries with a set of different compilation flags in order to cover as many real-world scenarios as possible. `Ramblr` successfully reassembles most of the binaries, which is an improvement over the state-of-the-art approach. It should be noted that our reassembling procedure yields no execution overhead and no size expansion.

I. Introduction

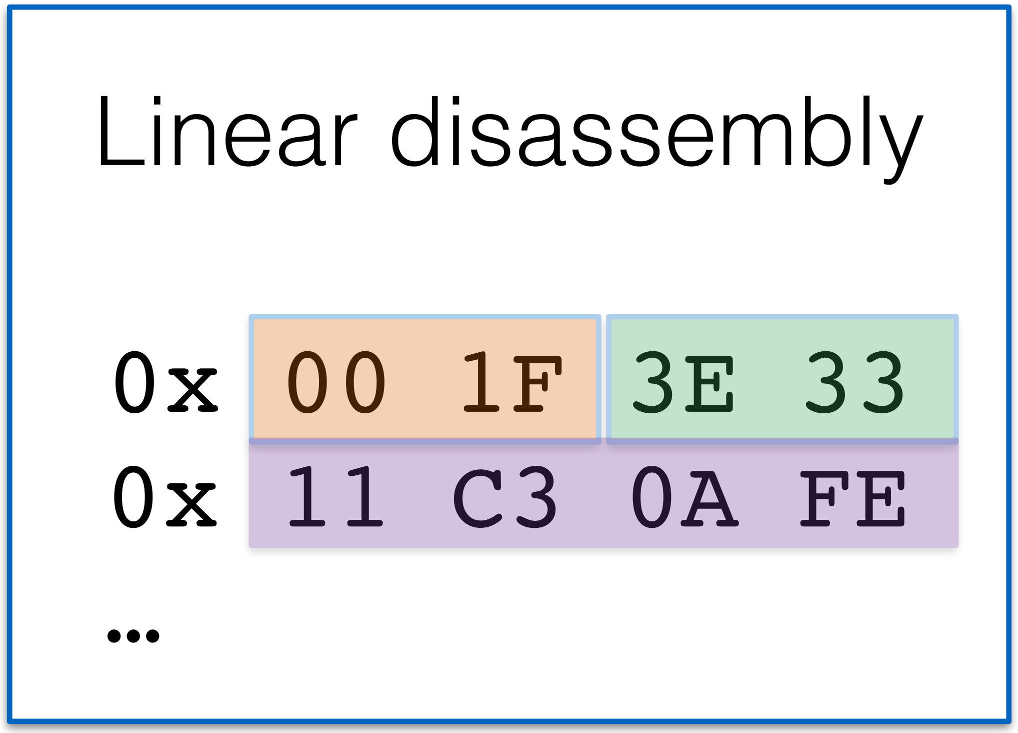
However, when source code is *absent*, such as in the case of proprietary software, the problem is much more complex. If the user of the software is unwilling to wait for the vendor to ship a new binary (or if the vendor no longer exists), the only option is to patch the binary directly.

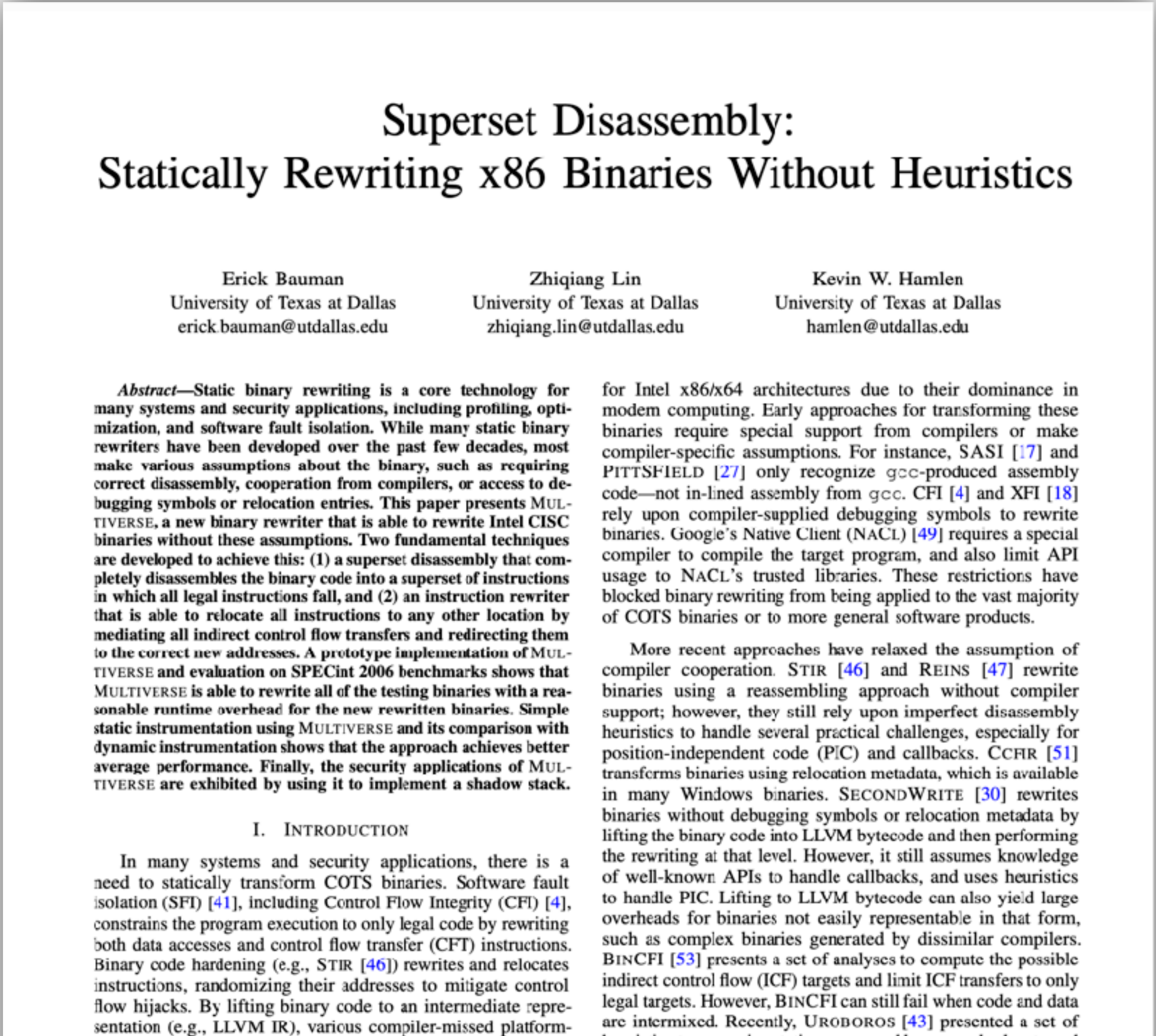
Patching binary code introduces challenges not present when patching source code. When a patch is applied at the source code level, the compiler will redo the process of arranging code and data in memory and handling links between them. In binary code, this is extremely difficult, since this linkage information is discarded by the compiler once finished. A performant binary patching process would need to rediscover the semantic meanings of different regions of program memory, and *reassemble* the program, redoing the compiler's arrangement while preserving cross-references among code and data. As a result of the difficulties inherent to this procedure, the patching of binary code is currently an ad-hoc process. Current work in the research community either makes unrealistically strict assumptions, does not provide realistic functionality guarantees, or results in significant performance and/or memory overhead. Because of this, no tool currently exists that can automatically and reliably patch real-world binary software.

Analysis-Directed Disassembly

- Most disassembly uses some local analyses
    - Intraprocedural control-flow, value set analysis, …
- Is every disassembler simply a poorly-designed abstract interpreter? Probably not. But maybe kind of?
- Common binary analysis structure: use program analysis to lift binary up to some arbitrarily-high-level IR
    - Common to instrumentation, RE, malware analysis, …

- Why is disassembly (of stripped x86-64 binaries) so hard?
  - No entrypoints (no *calling convention, even*)
  - Byte alignment
- Linear disassembly: start here, keep going
  - Lots of junk in `.text`





Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics

Erick Bauman
University of Texas at Dallas
erick.bauman@utdallas.edu

Zhiqiang Lin
University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Kevin W. Hamlen
University of Texas at Dallas
hamlen@utdallas.edu

***Abstract*—Static binary rewriting is a core technology for many systems and security applications, including profiling, optimization, and software fault isolation. While many static binary rewriters have been developed over the past few decades, most make various assumptions about the binary, such as requiring correct disassembly, cooperation from compilers, or access to debugging symbols or relocation entries. This paper presents MULTIVERSE, a new binary rewriter that is able to rewrite Intel CISC binaries without these assumptions. Two fundamental techniques are developed to achieve this: (1) a superset disassembly that completely disassembles the binary code into a superset of instructions in which all legal instructions fall, and (2) an instruction rewriter that is able to relocate all instructions to any other location by mediating all indirect control flow transfers and redirecting them to the correct new addresses. A prototype implementation of MULTIVERSE and evaluation on SPECint 2006 benchmarks shows that MULTIVERSE is able to rewrite all of the testing binaries with a reasonable runtime overhead for the new rewritten binaries. Simple static instrumentation using MULTIVERSE and its comparison with dynamic instrumentation shows that the approach achieves better average performance. Finally, the security applications of MULTIVERSE are exhibited by using it to implement a shadow stack.**

I. INTRODUCTION

In many systems and security applications, there is a need to statically transform COTS binaries. Software fault isolation (SFI) [41], including Control Flow Integrity (CFI) [4], constrains the program execution to only legal code by rewriting both data accesses and control flow transfer (CFT) instructions. Binary code hardening (e.g., STIR [46]) rewrites and relocates instructions, randomizing their addresses to mitigate control flow hijacks. By lifting binary code to an intermediate representation (e.g., LLVM IR), various compiler-missed platform-

for Intel x86/x64 architectures due to their dominance in modern computing. Early approaches for transforming these binaries require special support from compilers or make compiler-specific assumptions. For instance, SASI [17] and PITTSFIELD [27] only recognize gcc-produced assembly code—not in-lined assembly from gcc. CFI [4] and XFI [18] rely upon compiler-supplied debugging symbols to rewrite binaries. Google's Native Client (NACL) [49] requires a special compiler to compile the target program, and also limit API usage to NACL's trusted libraries. These restrictions have blocked binary rewriting from being applied to the vast majority of COTS binaries or to more general software products.

More recent approaches have relaxed the assumption of compiler cooperation. STIR [46] and REINS [47] rewrite binaries using a reassembling approach without compiler support; however, they still rely upon imperfect disassembly heuristics to handle several practical challenges, especially for position-independent code (PIC) and callbacks. CCFIR [51] transforms binaries using relocation metadata, which is available in many Windows binaries. SECONDWRITE [30] rewrites binaries without debugging symbols or relocation metadata by lifting the binary code into LLVM bytecode and then performing the rewriting at that level. However, it still assumes knowledge of well-known APIs to handle callbacks, and uses heuristics to handle PIC. Lifting to LLVM bytecode can also yield large overheads for binaries not easily representable in that form, such as complex binaries generated by dissimilar compilers. BINCFI [53] presents a set of analyses to compute the possible indirect control flow (ICF) targets and limit ICF transfers to only legal targets. However, BINCFI can still fail when code and data are intermixed. Recently, UROBOROS [43] presented a set of

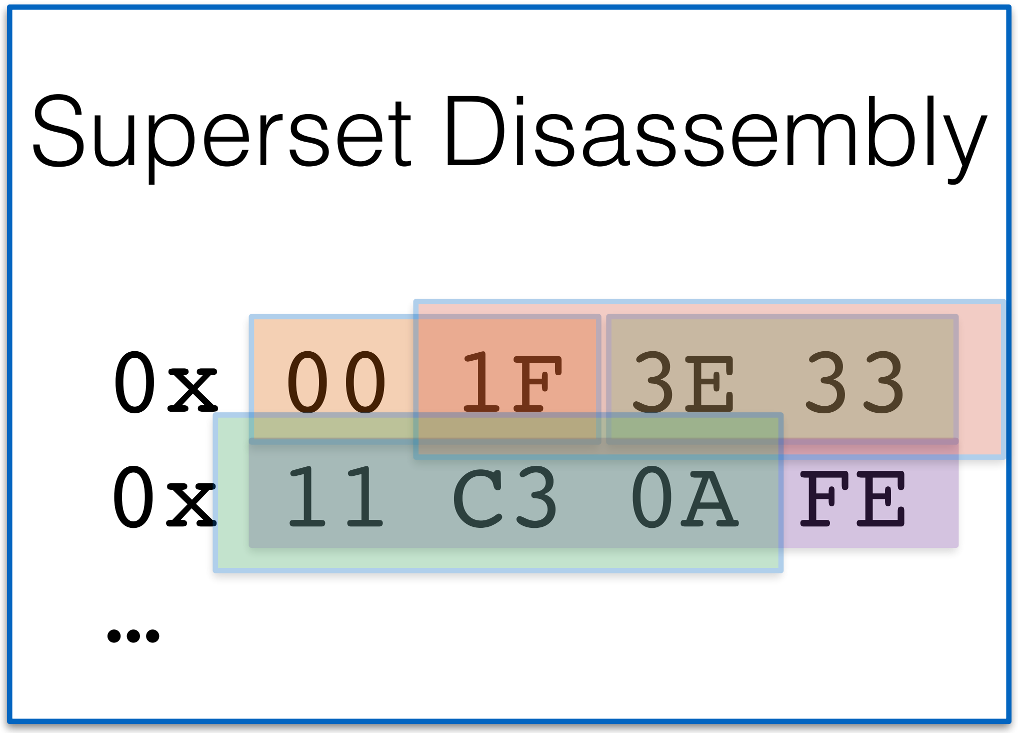
- Disassemble starting from *every possible instruction*

- Calculate a **superset** of disassemblies

- Derive analyses over various disassemblies

- Use heuristics to choose the "best" one

- **Lots** of computation, but *inherent parallelism* at analysis level



- **Problem**: all RE frameworks are slow, when used for superlinear

- **Design challenge**: most RE tools (IDA Pro, Ghidra, Radare2, BAP, …) are truly driven by lightweight static analyses around an MVC-type architecture

    - Some of these better than others (BAP [Brumley et al.])

    - But none uses truly modern analysis construction methodology

- **Usability challenge**: RE tools have very limited ability to be directed by the user

    - "I know it's using this nonstandard calling convention *here* (points to `0xFFEACA13`)…"

Datalog Disassembly

Antonio Flores-Montoya
GrammaTech, Inc.
afloresmontoya@grammatech.com

Eric Schulte
GrammaTech, Inc.
eschulte@grammatech.com

Abstract

Disassembly is fundamental to binary analysis and rewriting. We present a novel disassembly technique that takes a stripped binary and produces reassembleable assembly code. The resulting assembly code has accurate symbolic information, providing cross-references for analysis and to enable adjustment of code and data pointers to accommodate rewriting. Our technique features multiple static analyses and heuristics in a combined Datalog implementation. We argue that Datalog's inference process is particularly well suited for disassembly and the required analyses. Our implementation and experiments support this claim. We have implemented our approach into an open-source tool called `Ddisasm`. In extensive experiments in which we rewrite thousands of x64 binaries we find `Ddisasm` is both faster and more accurate than the current state-of-the-art binary reassembling tool, `Ramblr`.

1 Introduction

Software is increasingly ubiquitous and the identification and mitigation of software vulnerabilities is increasingly essential to the functioning of modern society. In many cases—e.g., COTS or legacy binaries, libraries, and drivers—source code is not available so identification and mitigation requires binary analysis and rewriting. Many disassemblers [9,10,23,31,36,56,57,59], analysis frameworks [4,7,12,19,25–27,29,39,48], rewriting frameworks [9,16,17,32,33,52,55,58,63], and reassembling tools [36,56,57] have been developed to support this need. Many applications depend on these tools including

Instruction boundaries Recovering where instructions start and end can be challenging especially in architectures such as x64 that have variable length instructions, dense instruction sets[1], and sometimes interleave code and data. This problem is also referred as *content classification*.

Symbolization information In binaries, there is no distinction between a number that represents a literal and a reference that points to a location in the code or data. If we modify a binary—e.g., by moving a block of code—all references pointing to that block, and to all of the subsequently shifted blocks, have to be updated. On the other hand, literals, even if they coincide with the address of a block, have to remain unchanged. This problem is also referred to as *Literal Reference Disambiguation*.

We have developed a disassembler that infers precise information for both questions and thus generates reassembleable assembly for a large variety of programs. These problems are not solvable in general so our approach leverages a combination of static program analysis and heuristics derived from empirical analysis of common compiler and assembler idioms. The static analysis, heuristics, and their combination are implemented in Datalog. Datalog is a declarative language that can be used to express dataflow analyses very concisely [50] and it has recently gained attention with the appearance of engines such as Soufflé [28] that generate highly efficient parallel C++ code from a Datalog program. We argue that Datalog is so well suited to the implementation of a disassembler that it represents a qualitative change in what is possible in terms of accuracy and efficiency.

We can conceptualize disassembly as taking a series of de-

Datalog Disassembly

- Declarative rules specify a set of analyses used to ultimately derive superset disassembly
  - Instruction identification, basic blocks, local control-flow, interprocedural control-flow (approx), symbolization, …
- Modern Datalog solvers (Soufflé) ***fast***, compile to relational algebra (i.e., just a bunch of cache-friendly for loops)
  - Superset disassembly *still* slower than linear (by 10-100x)
- Details of fixed-point iteration abstracted away
  - Very small, clean implementation; analysis clearly explicated

```
//use just the constant in the jump
jump_table(EA,Initial_memory):-
    code(EA),
    indirect_jump(EA),
    arch.pointer_size(Pt_size),
    symbolic_operand(EA,1,Initial_memory,"data"),
    symbolic_data(Initial_memory,Pt_size,Dest_block),
    refined_block(Dest_block).
```

```
inter_procedural_jump(Src,Dest):-
    unconditional_jump(Src),
    direct_jump(Src,Dest),
    direct_call(OtherSrc,Dest),
    code(OtherSrc).
```

- Calculate superset of disassemblies via Datalog
- Soufflé allows exploiting (some degree of) data parallelism
- Use analysis to assign “score” to ground / explicate heuristics used for disassembly
    - Is this prone to malicious attacks? We don’t know!
    - Use ML to train these weights for better disassembly?

Reversi—Analysis-Directed RE

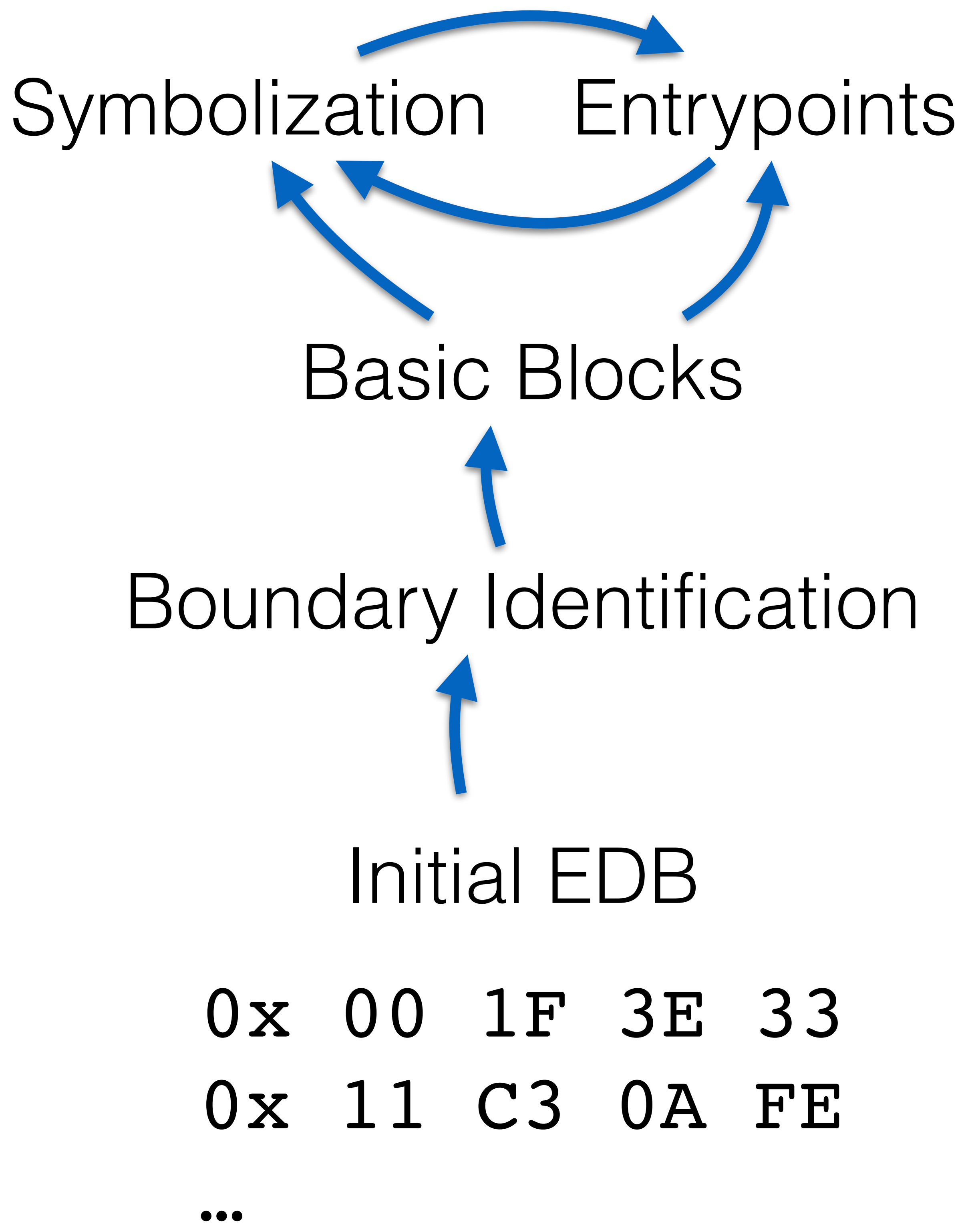
- Our Datalog-based RE framework

- RE, or (*equivalently*) binary rewriting, is tower of IRs, starting from binary, up to disassembly, the final source artifact, …

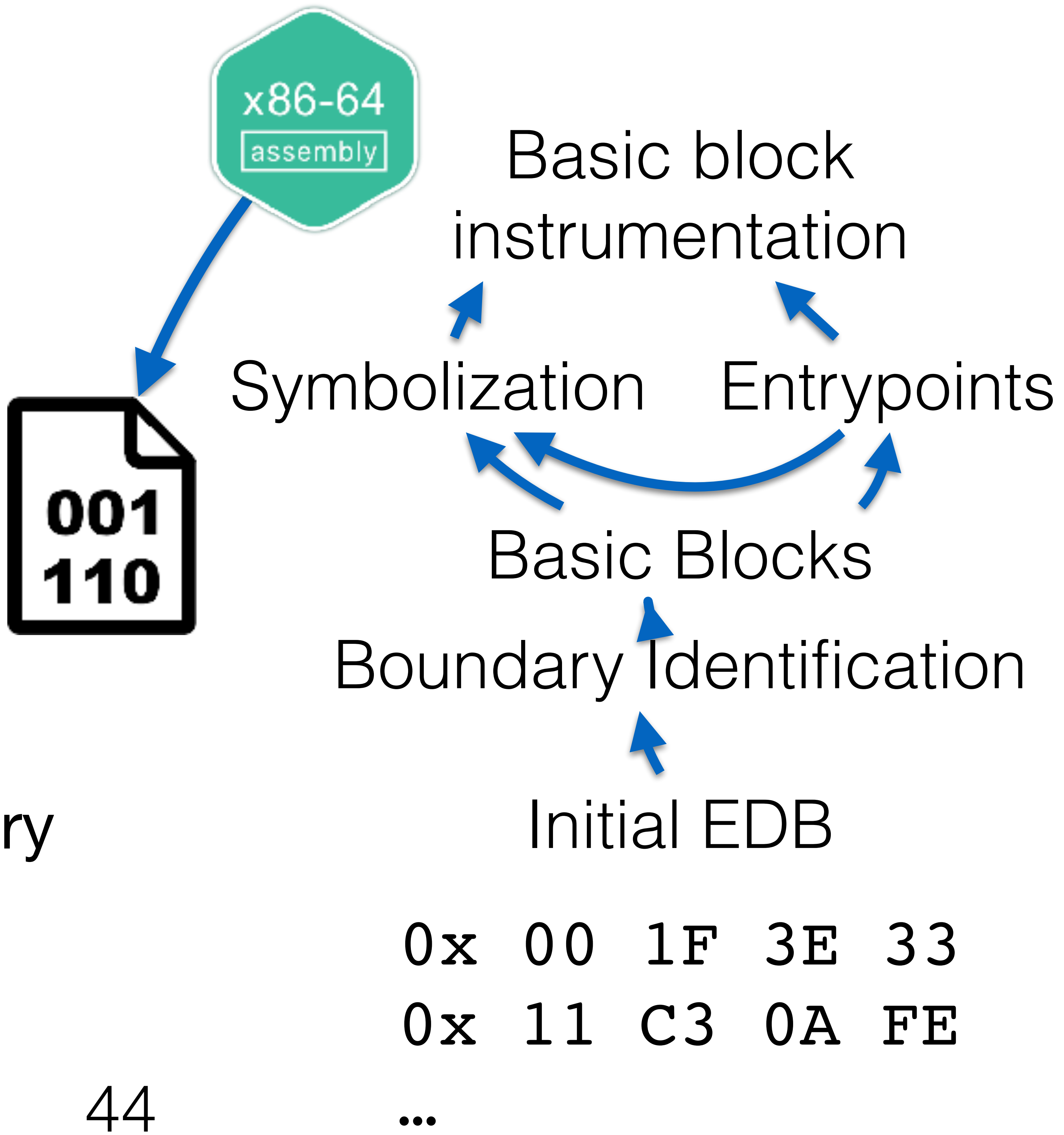
  - (Maybe source + assembly)

- Each step in hierarchy defined by declarative rules

  - Implemented as extension to ddisasm



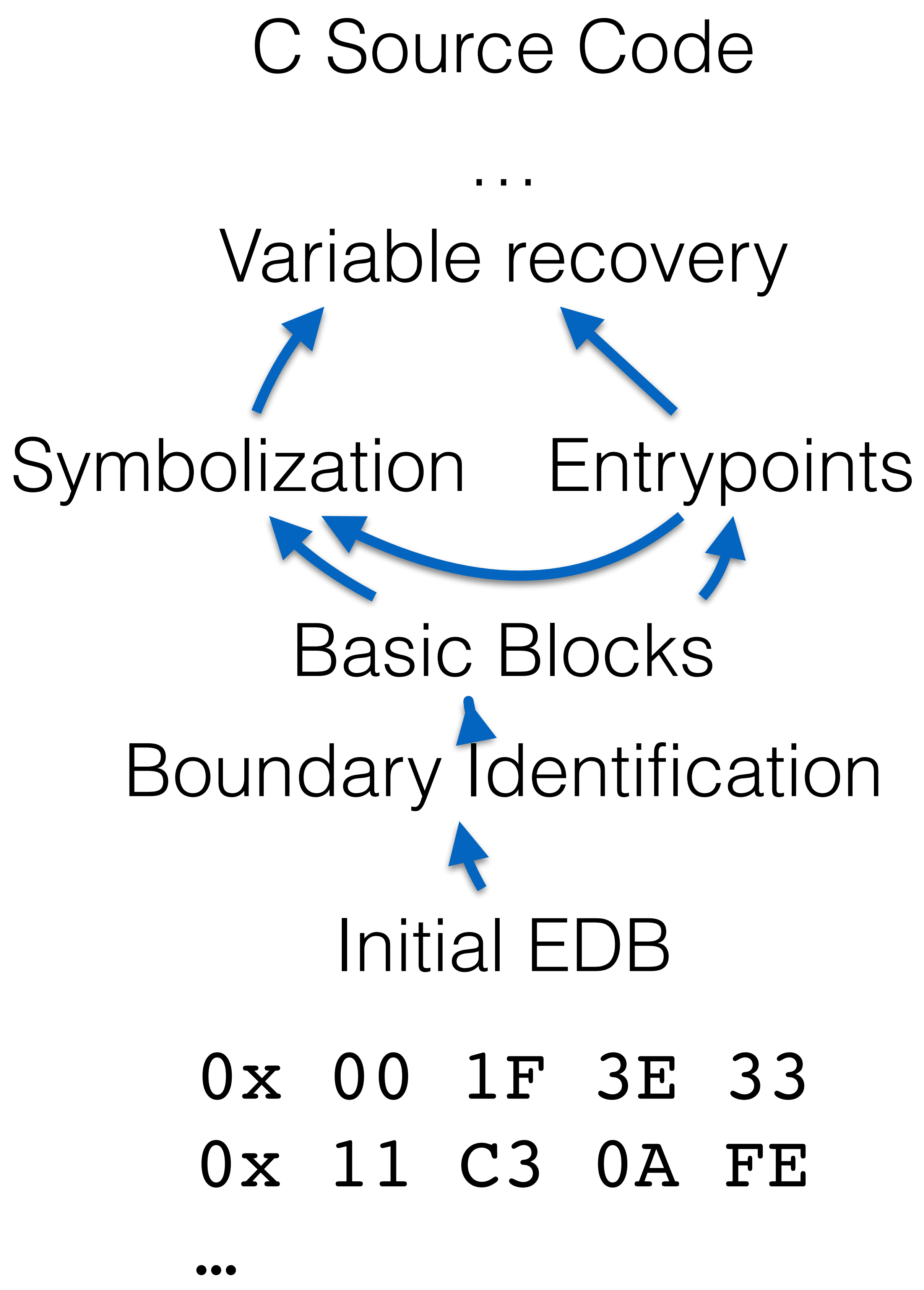
What should an RE tool output?



- Right now: modified assembly file, new binary

What should an RE tool output?

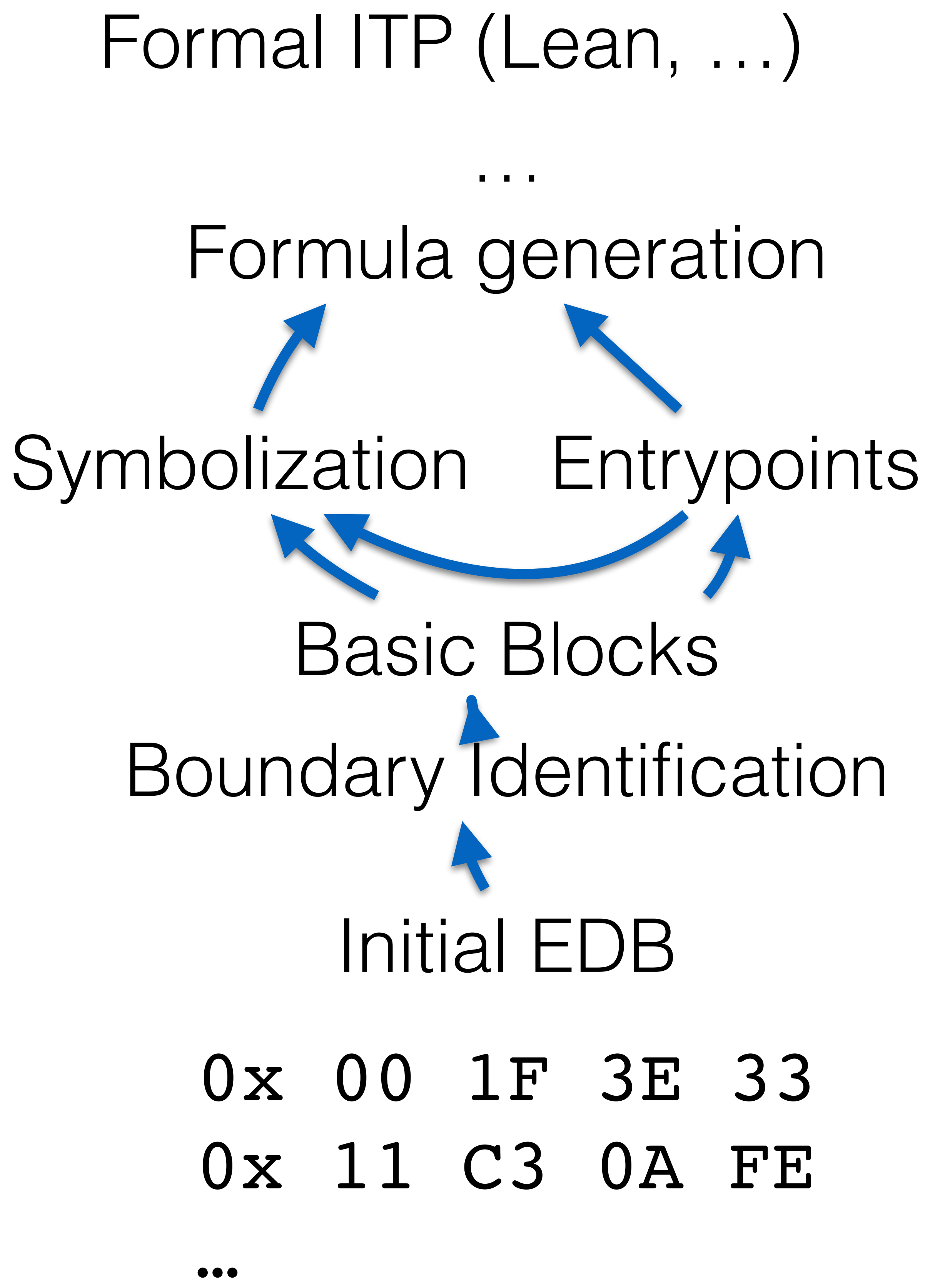
- Long term: a variety of backends
    - Decompiled Source



What should an RE tool output?

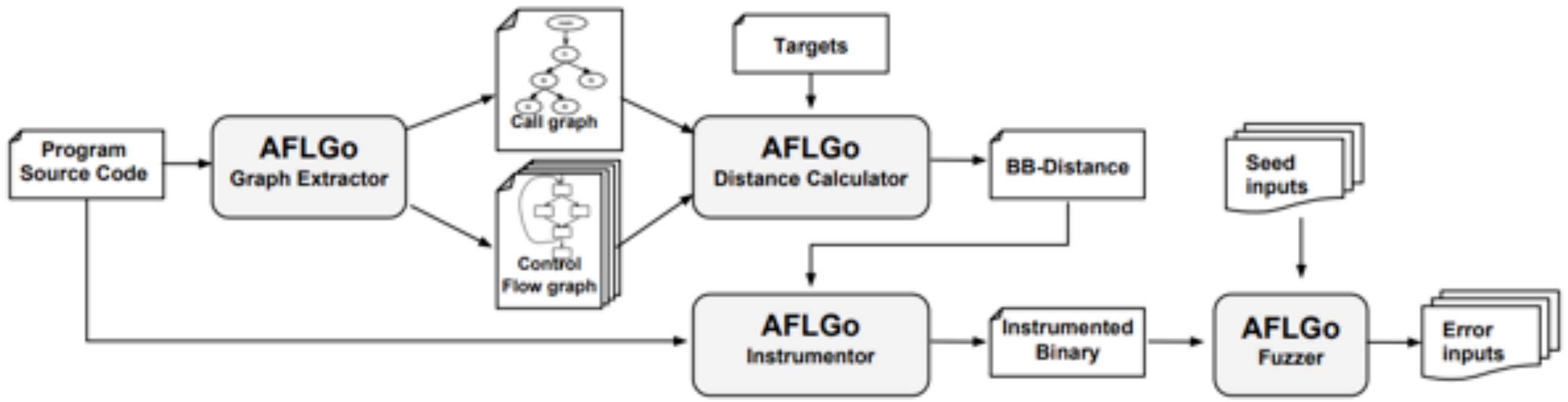
- Long term: a variety of backends

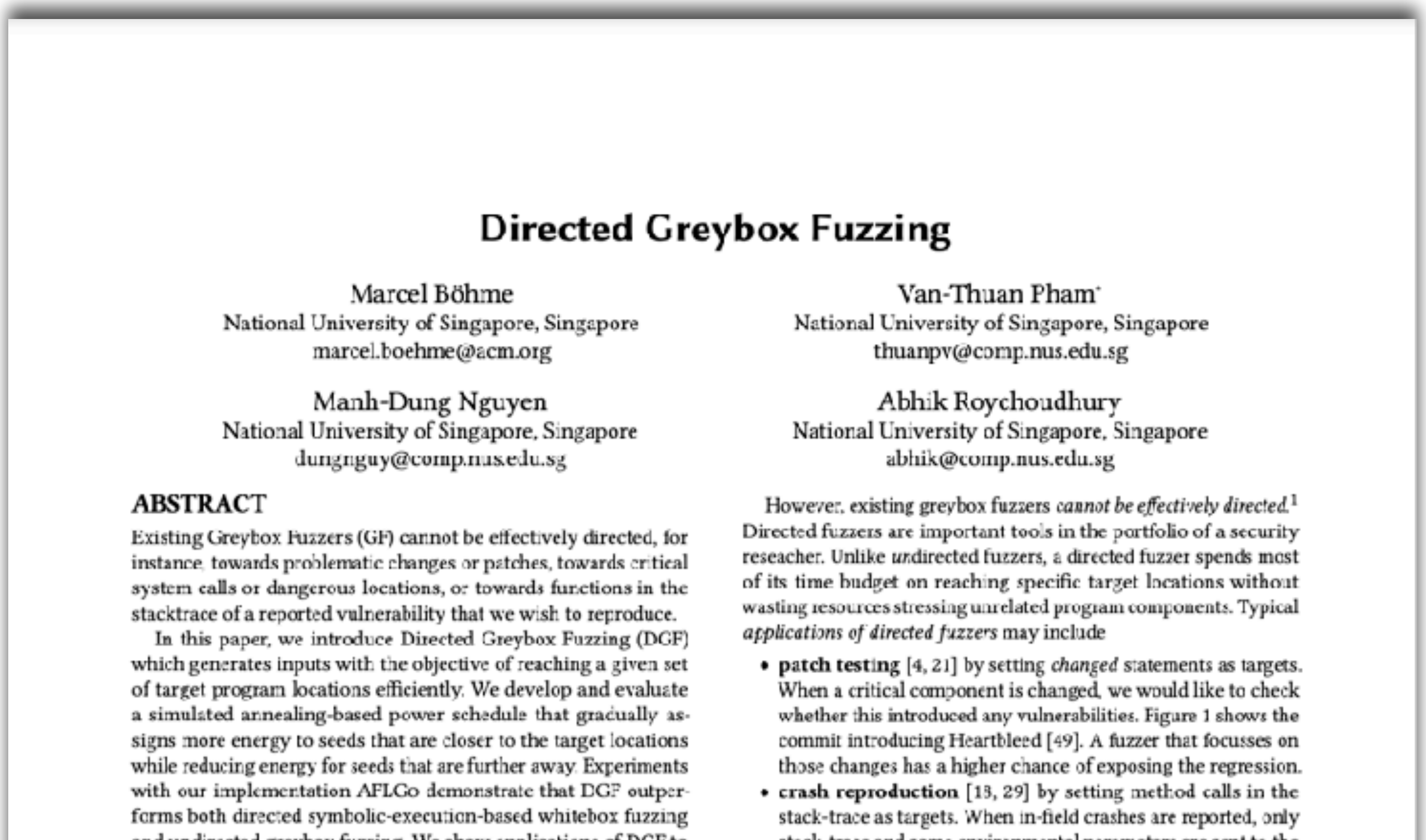
    - ITP term language (Lean, Gallina, …) alongside formal ISA spec



Currently—Fuzzing Instrumentation

- Fuzzing tools, advances in fuzzing, all essentially use LLVM
    - Everyone assumes you can do this for binaries
- Modern, coverage-guided fuzzing tools instrument *control-flow edges* to record coverage in bitmap
    - Bitmap fits in cache, always resident, instrumentation adds very little overhead—radically better throughput than SE
- Applying these tools to binaries has been a challenge





Directed Greybox Fuzzing

Marcel Böhme
National University of Singapore, Singapore
marcel.boehme@acm.org

Van-Thuan Pham*
National University of Singapore, Singapore
thuanpv@comp.nus.edu.sg

Manh-Dung Nguyen
National University of Singapore, Singapore
dungnguy@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Existing Greybox Fuzzers (GF) cannot be effectively directed, for instance, towards problematic changes or patches, towards critical system calls or dangerous locations, or towards functions in the stacktrace of a reported vulnerability that we wish to reproduce.

In this paper, we introduce Directed Greybox Fuzzing (DGF) which generates inputs with the objective of reaching a given set of target program locations efficiently. We develop and evaluate a simulated annealing-based power schedule that gradually assigns more energy to seeds that are closer to the target locations while reducing energy for seeds that are further away. Experiments with our implementation AFLGo demonstrate that DGF outperforms both directed symbolic-execution-based whitebox fuzzing

However, existing greybox fuzzers *cannot be effectively directed*.[1] Directed fuzzers are important tools in the portfolio of a security reseacher. Unlike undirected fuzzers, a directed fuzzer spends most of its time budget on reaching specific target locations without wasting resources stressing unrelated program components. Typical *applications of directed fuzzers* may include

- **patch testing** [4, 21] by setting *changed* statements as targets. When a critical component is changed, we would like to check whether this introduced any vulnerabilities. Figure 1 shows the commit introducing Heartbleed [49]. A fuzzer that focusses on those changes has a higher chance of exposing the regression.
- **crash reproduction** [13, 29] by setting method calls in the stack-trace as targets. When in-field crashes are reported, only

Our Approach

- Declarative API to define how instrumentation should occur
  - Based on the results of Datalog disassembly
- Output of instrumentation is reassembleable assembly
- Reversi built using ddisasm, defines how to extend binary
  - Avoids the general pain points of binary rewriting—lots of information is available to you from ddisasm!

```
leaq -(128+24)(%rsp), %rsp
movq %rdx,  0(%rsp)
movq %rcx,  8(%rsp)
movq %rax, 16(%rsp)
movq {unique_mark}, %rcx
call __afl_maybe_log
movq 16(%rsp), %rax
movq  8(%rsp), %rcx
movq  0(%rsp), %rdx
leaq (128+24)(%rsp), %rsp
leaq -(128+24)(%rsp), %rsp
movq %rdx,  0(%rsp)
movq %rcx,  8(%rsp)
movq %rax, 16(%rsp)
movq {unique_mark}, %rcx
call __afl_maybe_log
movq 16(%rsp), %rax
movq  8(%rsp), %rcx
movq  0(%rsp), %rdx
leaq (128+24)(%rsp), %rsp
```

```
// declare asm snippet
asm_snippet("trampoline", "trampoline_64.s").

// instrument trampoline at the start of each basic block
insert_at(block_start, "trampoline", 1),
replace_placeholder_at(block_start, "trampoline",
                       "unique_mark", block_mark) :-
    map_size(m_size),
    text_block(block_start),
    !lowest_block(block_start),
    block_mark = cat("$", to_string(@random32(m_size))).
```

```
leaq -(128+24)(%rsp), %rsp
movq %rdx,   0(%rsp)
movq %rcx,   8(%rsp)
movq %rax,  16(%rsp)
movq {unique_mark}, %rcx
call __afl_maybe_log
movq 16(%rsp), %rax
movq  8(%rsp), %rcx
movq  0(%rsp), %rdx
leaq (128+24)(%rsp), %rsp
```

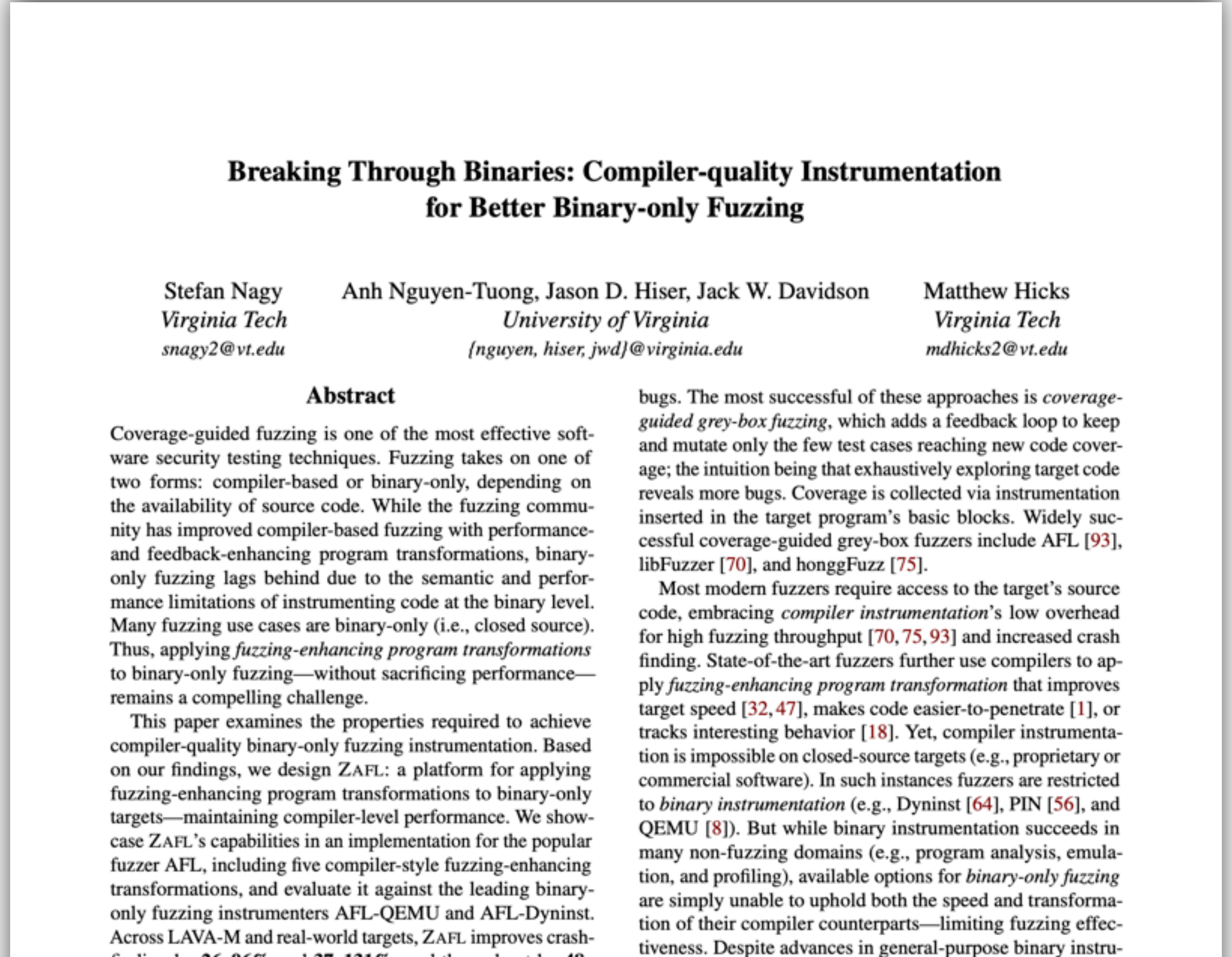
```
// declare asm snippet
asm_snippet("trampoline", "trampoline_64.s").

// instrument trampoline at the start of each basic block
insert_at(block_start, "trampoline", 1),
replace_placeholder_at(block_start, "trampoline",
                       "unique_mark", block_mark) :-
    map_size(m_size),
    text_block(block_start),
    !lowest_block(block_start),
    block_mark = cat("$", to_string(@random32(m_size))).
```

Initial Results

- Replicated AFL's LLVM-based instrumentation pass
- Reversi *works* as fast as AFL's normal LLVM-based instrumentation
    - 3 different apps, 5 runs for 1 hr each—~same # of crashes / paths for each run
    - We instrument *fewer* edges due to increased precision
- Reversi **much faster** than the QEMU-based binary backend
    - Up to 2-5x improvement (QEMU is very slow)
    - IOW: already doing something new vs. state of the art

Replicating these results, but much more pluggable



Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing

Stefan Nagy
Virginia Tech
snagy2@vt.edu

Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson
University of Virginia
{nguyen, hiser, jwd}@virginia.edu

Matthew Hicks
Virginia Tech
mdhicks2@vt.edu

Abstract

Coverage-guided fuzzing is one of the most effective software security testing techniques. Fuzzing takes on one of two forms: compiler-based or binary-only, depending on the availability of source code. While the fuzzing community has improved compiler-based fuzzing with performance- and feedback-enhancing program transformations, binary-only fuzzing lags behind due to the semantic and performance limitations of instrumenting code at the binary level. Many fuzzing use cases are binary-only (i.e., closed source). Thus, applying *fuzzing-enhancing program transformations* to binary-only fuzzing—without sacrificing performance—remains a compelling challenge.

This paper examines the properties required to achieve compiler-quality binary-only fuzzing instrumentation. Based on our findings, we design ZAFL: a platform for applying fuzzing-enhancing program transformations to binary-only targets—maintaining compiler-level performance. We showcase ZAFL's capabilities in an implementation for the popular fuzzer AFL, including five compiler-style fuzzing-enhancing transformations, and evaluate it against the leading binary-only fuzzing instrumenters AFL-QEMU and AFL-Dyninst. Across LAVA-M and real-world targets, ZAFL improves crash-

bugs. The most successful of these approaches is *coverage-guided grey-box fuzzing*, which adds a feedback loop to keep and mutate only the few test cases reaching new code coverage; the intuition being that exhaustively exploring target code reveals more bugs. Coverage is collected via instrumentation inserted in the target program's basic blocks. Widely successful coverage-guided grey-box fuzzers include AFL [93], libFuzzer [70], and honggFuzz [75].

Most modern fuzzers require access to the target's source code, embracing *compiler instrumentation*'s low overhead for high fuzzing throughput [70, 75, 93] and increased crash finding. State-of-the-art fuzzers further use compilers to apply *fuzzing-enhancing program transformation* that improves target speed [32, 47], makes code easier-to-penetrate [1], or tracks interesting behavior [18]. Yet, compiler instrumentation is impossible on closed-source targets (e.g., proprietary or commercial software). In such instances fuzzers are restricted to *binary instrumentation* (e.g., Dyninst [64], PIN [56], and QEMU [8]). But while binary instrumentation succeeds in many non-fuzzing domains (e.g., program analysis, emulation, and profiling), available options for *binary-only fuzzing* are simply unable to uphold both the speed and transformation of their compiler counterparts—limiting fuzzing effectiveness. Despite advances in general-purpose binary instru-

Thanks!

- Practical and theoretically interesting advances to be had in taking a PL perspective on binary analysis
- Most binary analysis tools are ad-hoc static analyses
    - This is where the real pain points are in these tools!
- Declarative binary analyses open many doors
    - User-directed decompilation, declarative fuzzing, etc…
- Assemblage: reproducibly build huge binary corpuses
- Drew: declarative binary rewriting