BinRec: Dynamic Binary Lifting and Recompilation

Anil Altinay∗, **Joseph Nash∗,** Taddeus Kroes∗

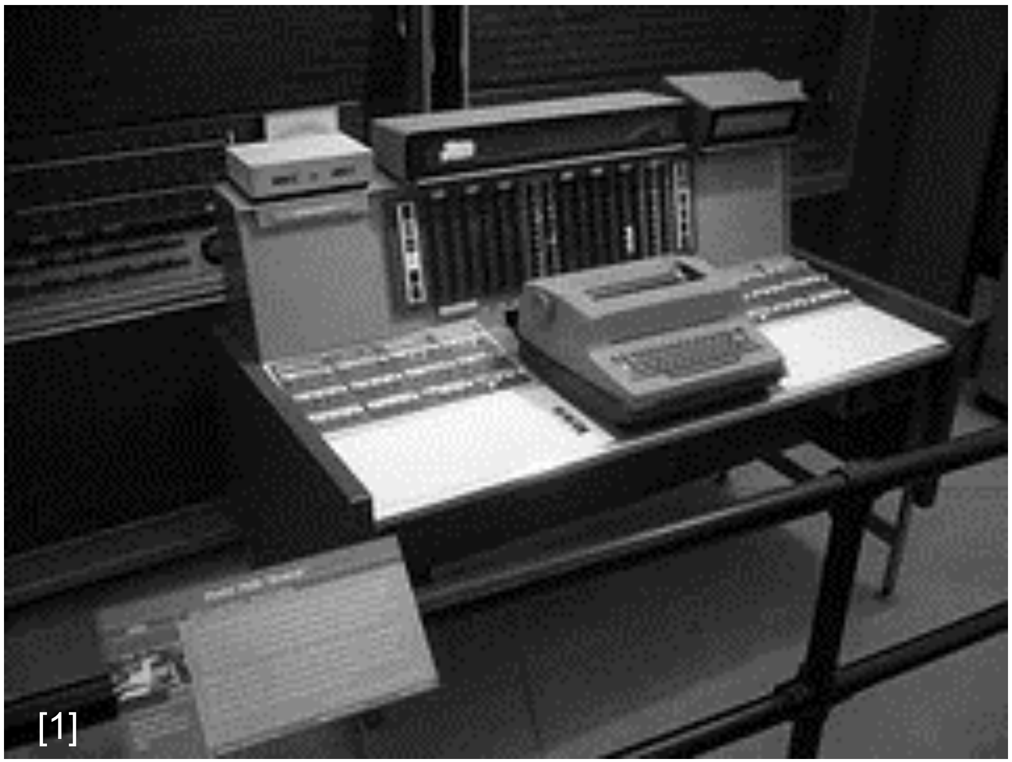
Prabhu Rajasekaran, Dixin Zhou,

Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert,

Cristiano Giuffrida, Herbert Bos, Michael Franz

∗Equal Contribution Joint-First Authors

Legacy Binaries Need Help

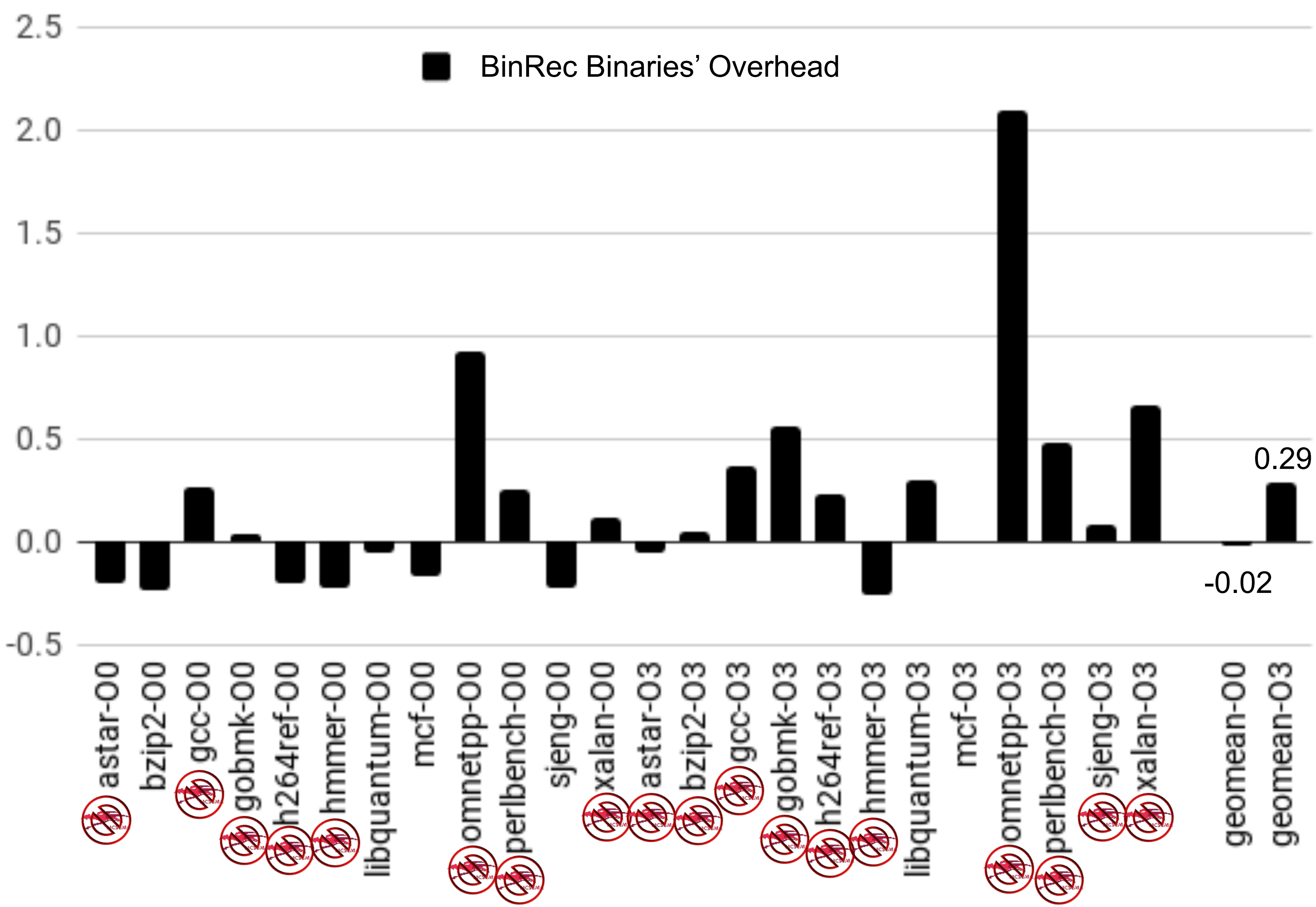


- Source code or toolchain has been lost
- Microsoft patched CVE-2017-11882 in Equation Editor
- Binary Rewriting to patch, reoptimize, instrument, or harden binaries

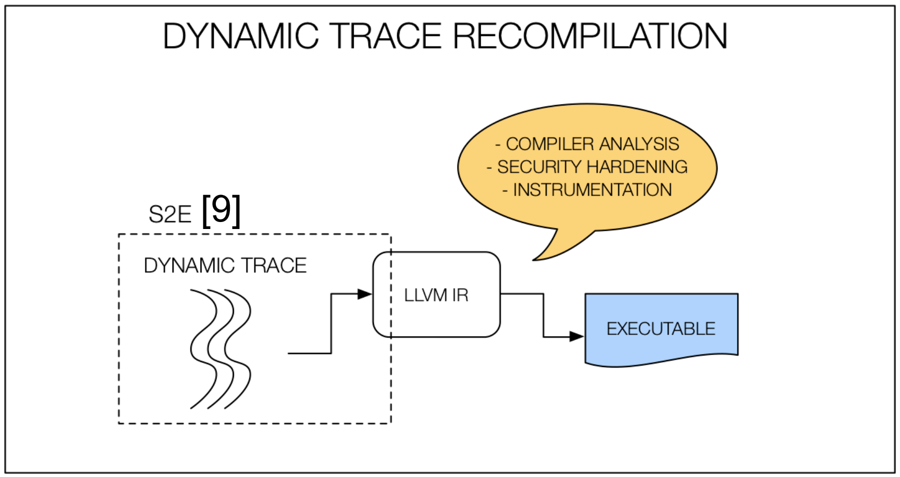
Limitations of Static Rewriting

- 5 challenges for static binary rewriting
  - Code vs Data Separation
  - Indirect Control Flow Resolution
  - Ill-formed Code
  - Obfuscation
  - External Entry Points
- Static approaches use **heuristics** since they can't solve these challenges in a principled way
- Produce rewritten binaries with **poor performance**, especially with instrumentation
- Require **re-implementing** well known analyses within every framework

BinRec vs McSema[6]



BinRec Framework



Highlights

- Lift binaries to LLVM IR
- Enable off-the-shelf compiler transformations
  - Safe Stack, ASAN, Optimizations, De-obfuscation, CFI
- Lift and run all C/C++ benchmarks in SPEC CINT 2006
- Better performing than existing lifting frameworks
  - Rev.ng[13] : 2.25x (static linked)
  - Multiverse[7] :1.60x (w/o instrumentation)
  - McSema[6] : >2x (only 4 binaries)
  - BinRec :1.29x

Leveraging Dynamic Traces to Overcome Static Rewriting Challenges

Code vs Data

- A statically unsolvable problem (Horspool and Marovac [3])
- Solution:
  - Copy of original program in case of inlined code and data as in prior work [10,11]
  - Dynamically observe the use of ambiguous values
  - Never accidentally disassemble data as code.
- libjpeg example [12]

Code vs Data in libjpeg

```c
void callback_func(j_common_ptr cinfo) {
  printf(".");
}

int main (int argc, char **argv) {
    struct jpeg_decompress_struct info; // jpeg info
    struct jpeg_progress_mgr  progress;
    ...
    // After some initialization code
    progress.progress_monitor = callback_func;
    progress.pass_limit = 0x8048860;
    progress.pass_counter = 0L;

    info.progress = &progress;
    jpeg_start_decompress( &info );

    char *data = (char *) malloc(dataSize);
    readData(info, data);
    ...
}
```

McSema mis-handles this case!

Callback function is stored in a struct

Constant is same as address of callback function

Code vs Data in libjpeg

```
void callback_func(j_common_ptr cinfo) {
  printf(".");
}

int main (int argc, char **argv) {
    struct jpeg_decompress_struct info; //jpeg info
    struct jpeg_progress_mgr  progress;
    ...
    //After some initialization code
    progress.progress_monitor = callback_func;
    progress.pass_limit = 0x8048860;
    progress.pass_counter = 0L;

    info.progress = &progress;
    jpeg_start_decompress( &info );

    char *data = (char *) malloc(dataSize);
    readData(info, data);
    ...
}
```

McSema mis-handles this case!

Callback function is stored in a struct

Constant is same as address of callback function

Code vs Data in libjpeg

```
void callback_func(j_common_ptr cinfo) {
  printf(".");
}

int main (int argc, char **argv) {
    struct jpeg_decompress_struct info; // jpeg info
    struct jpeg_progress_mgr  progress;
    ...
    // After some initialization code
    progress.progress_monitor = callback_func;
    progress.pass_limit = 0x8048860;
    progress.pass_counter = 0L;

    info.progress = &progress;
    jpeg_start_decompress( &info );

    char *data = (char *) malloc(dataSize);
    readData(info, data);
    ...
}
```

McSema mis-handles this case!

Callback function is stored in a struct

Constant is same as address of callback function

Indirect Control Flow

- Static approaches use heuristics with value set analysis
- BinRec records the exact target addresses of each indirect control flow

ret

Traces observed:
ret to A
ret to B

```
%pc = load i32, i32* @PC
switch  %pc, label %otherwise
        [  i32 &A, label %BasicBlock_A
           i32 &B, label %Basicblock_B  ]
```

External Entry Points: Callbacks

Binary Code

Callback function

```
int compare( const void* a, const void* b ) {
....
....
}

int main() {
    int arr[] = {5, 3, 1, -1};

    int size = sizeof arr / sizeof *arr;
    qsort( arr, size, sizeof( int ),
                  compare);
}
```

Passed to qsort function

Library Code

```
void qsort(void *base,
        size_t nel,
        size_t width,
        int (*compar)(const void *,
                      const void *))
{
        .....
        .....
        .....
        compare(arg1, arg2);
}
```

qsort invokes callback function

Support for External Entry Points

Problem: The callback function pointer still points to the original callback function

Library Code

```
void qsort(void *base,
        size_t nel,
        size_t width,
        int (*compar)(const void *,
                      const void *))
{
        .....
        .....
        .....
2       compare(arg1, arg2);
}
```

qsort invokes original callback function

Recovered Code

```
int compare_recovered( .... ) {
....
}

int main_recovered() {
....
qsort( ...., compare);      1
}
```

Support for External Entry Points

Problem: The callback function pointer still points to the original callback function

Recovered Code

```
int compare_recovered( …. ) {
….
}

int main_recovered() {
….
qsort( …., compare);    1
}
```

Library Code

```
void qsort(void *base,
       size_t nel,
       size_t width,
       int (*compar)(const void *,
                     const void *))
{
       …..
       …..
       …..
2      compare(arg1, arg2);
}
```

qsort invokes original callback function

Support for External Entry Points

- Option 1: statically link library code into the analysis region
  - Problem: High memory usage
- Option 2: update code pointers
  - Problem: Heuristics fail
- Option 3: create a lookup table
  - Problem: Performance degradation

Support for External Entry Points

Our Dynamic Approach

Original Code Region

```
compare: jmp compare_recovered
```

Use original address space as trampolines

Library Code

```
void qsort(void *base,
        size_t nel,
        size_t width,
        int (*compar)(const void *,
                      const void *))
{
        .....
        .....
        .....
        compare(arg1, arg2);
}
```

3.

Recovered Code

```
int compare_recovered( .... ) {
....
}

int main_recovered() {
....
qsort( ...., compare);
}
```

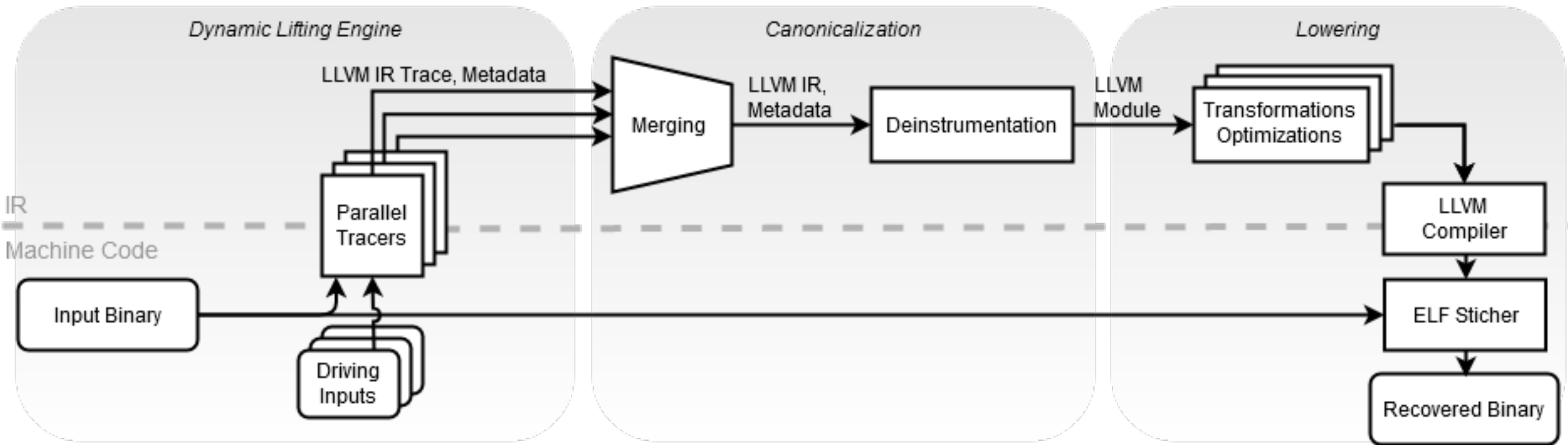
1.

2.

4.

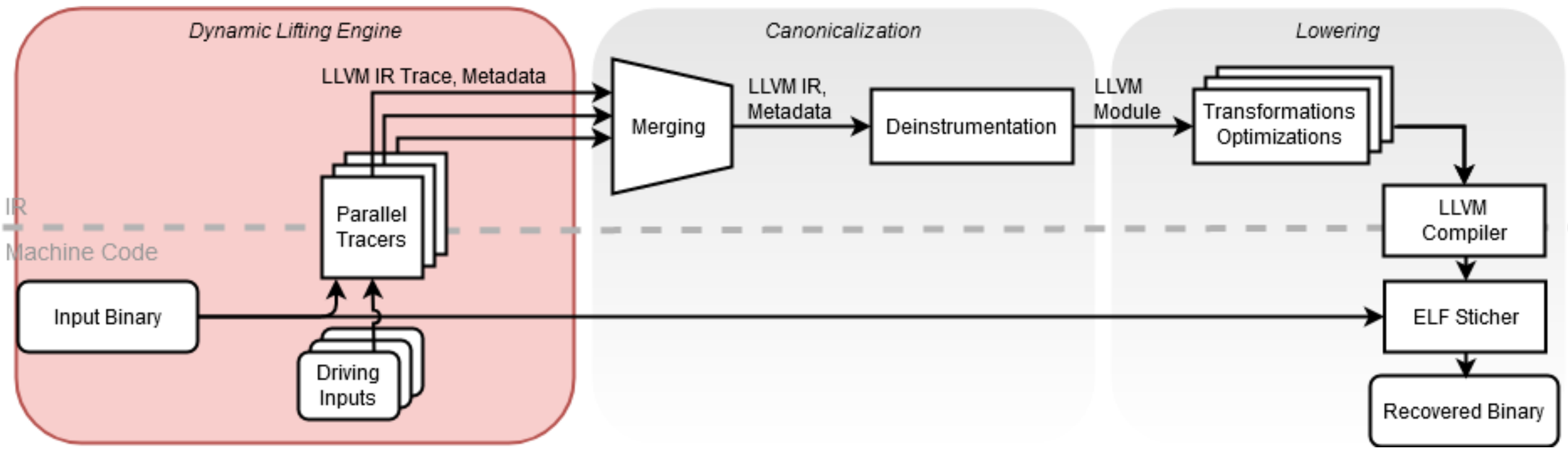
No need for arguments patching!

BinRec Architected for Coverage



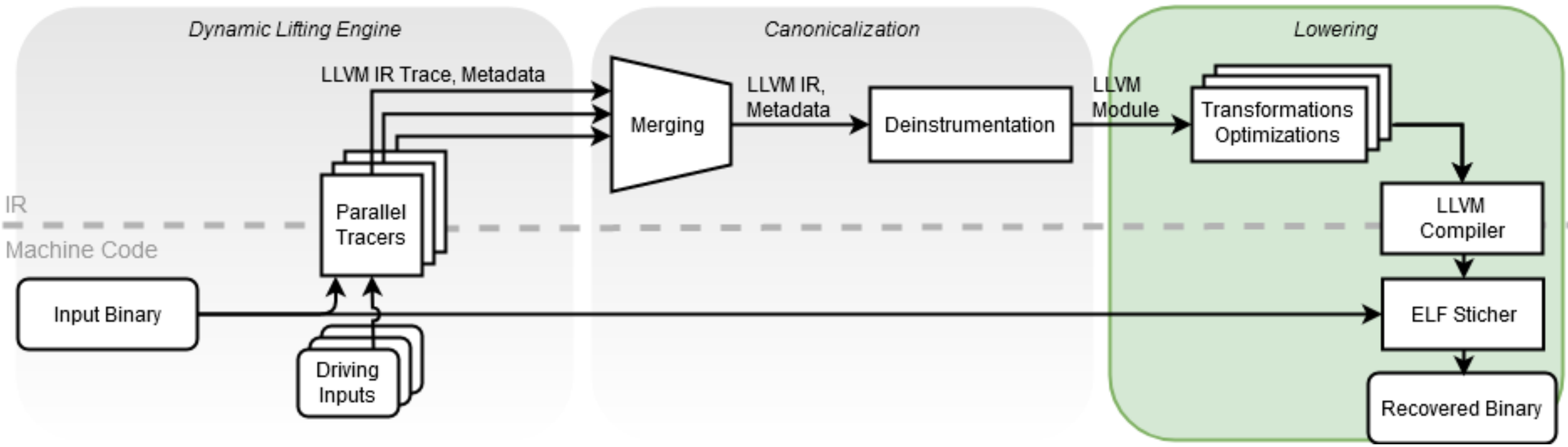
- Coverage for Dynamic Analysis
- Dynamic lifting engine efficiently covers paths of interest
- Installed handlers provides recovery and iterative improvement

BinRec Architected for Coverage



- Coverage for Dynamic Analysis
- Dynamic lifting engine efficiently covers paths of interest
- Installed handlers provides recovery and iterative improvement

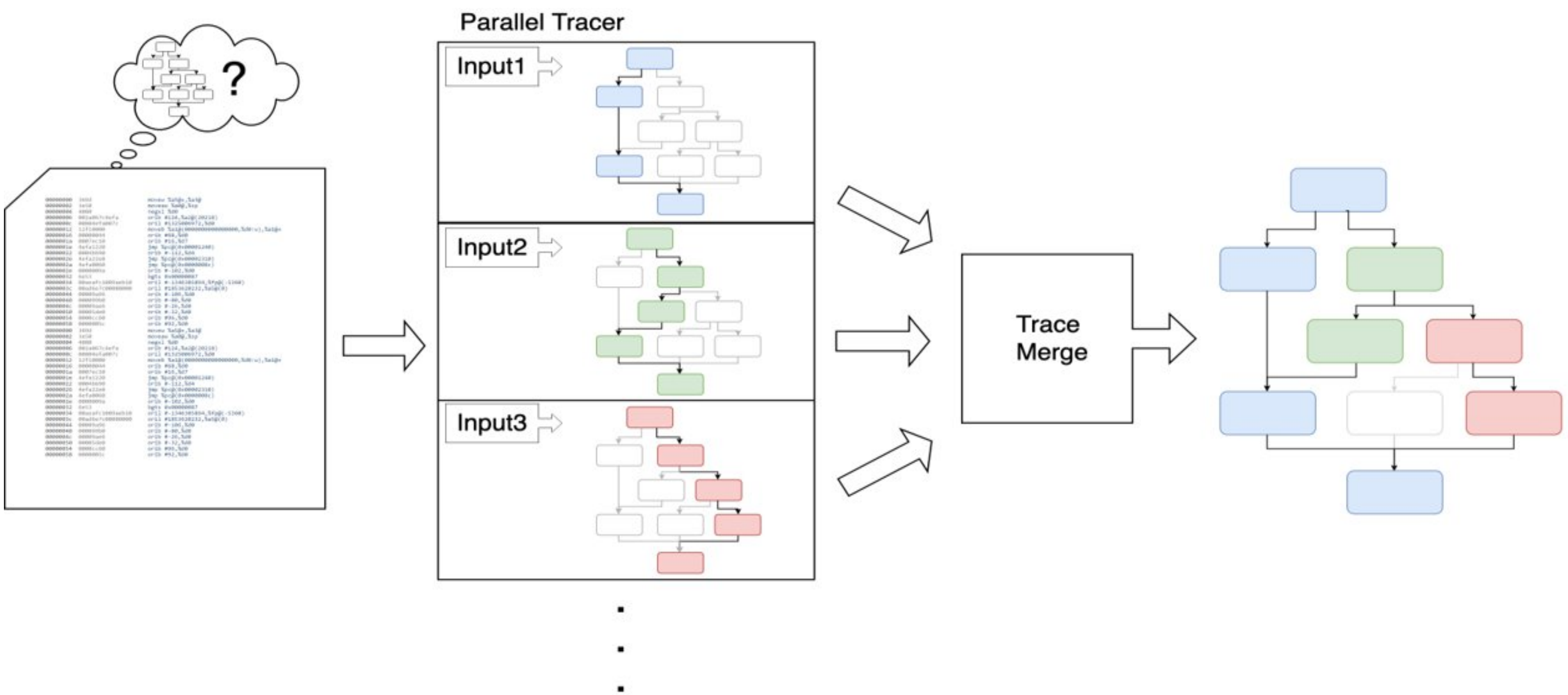
BinRec Architected for Coverage



- Coverage for Dynamic Analysis
- Dynamic lifting engine efficiently covers paths of interest
- Installed handlers provides recovery and iterative improvement

Multi-Trace Merging

- **Drive execution** - Trusted inputs, fuzzing, concolic execution
- **Build CFG –** Merge basic block boundaries, control flow edges

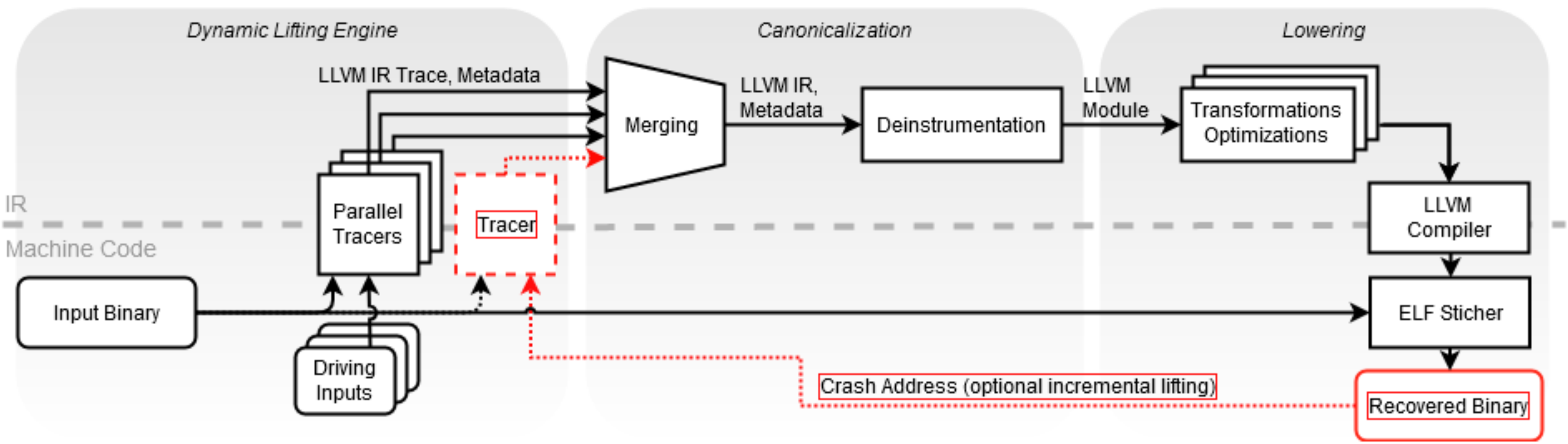


Configurable Pass Miss Handlers

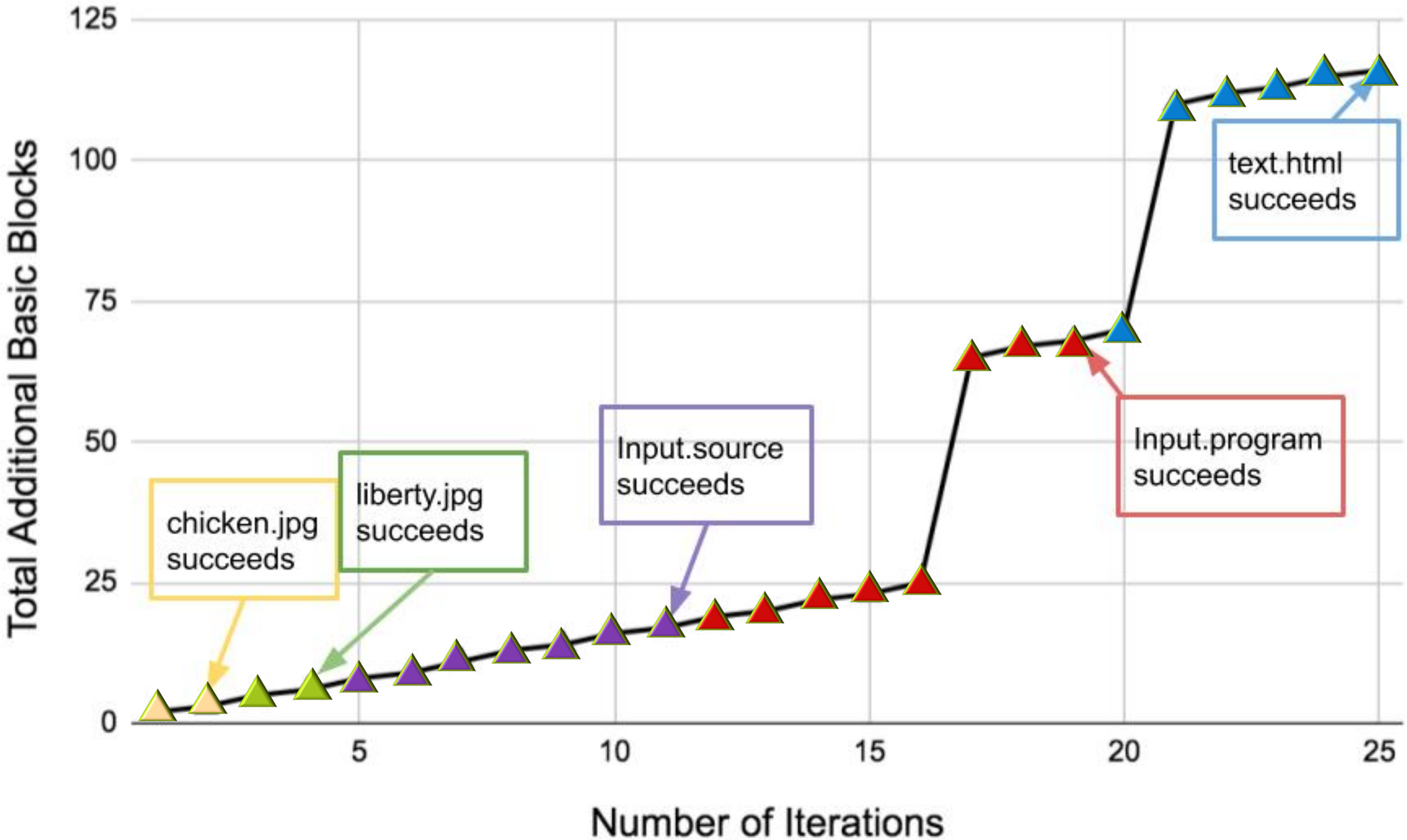
- Path Miss := instructions needed for the current workload were not observed in the initial lifting
- Path Miss Handlers are installed in every control flow transfer
  - Optimized Out
  - Report and Log
  - Fallback
  - Incremental Lifting

Path Miss Handler: Incremental Lifting

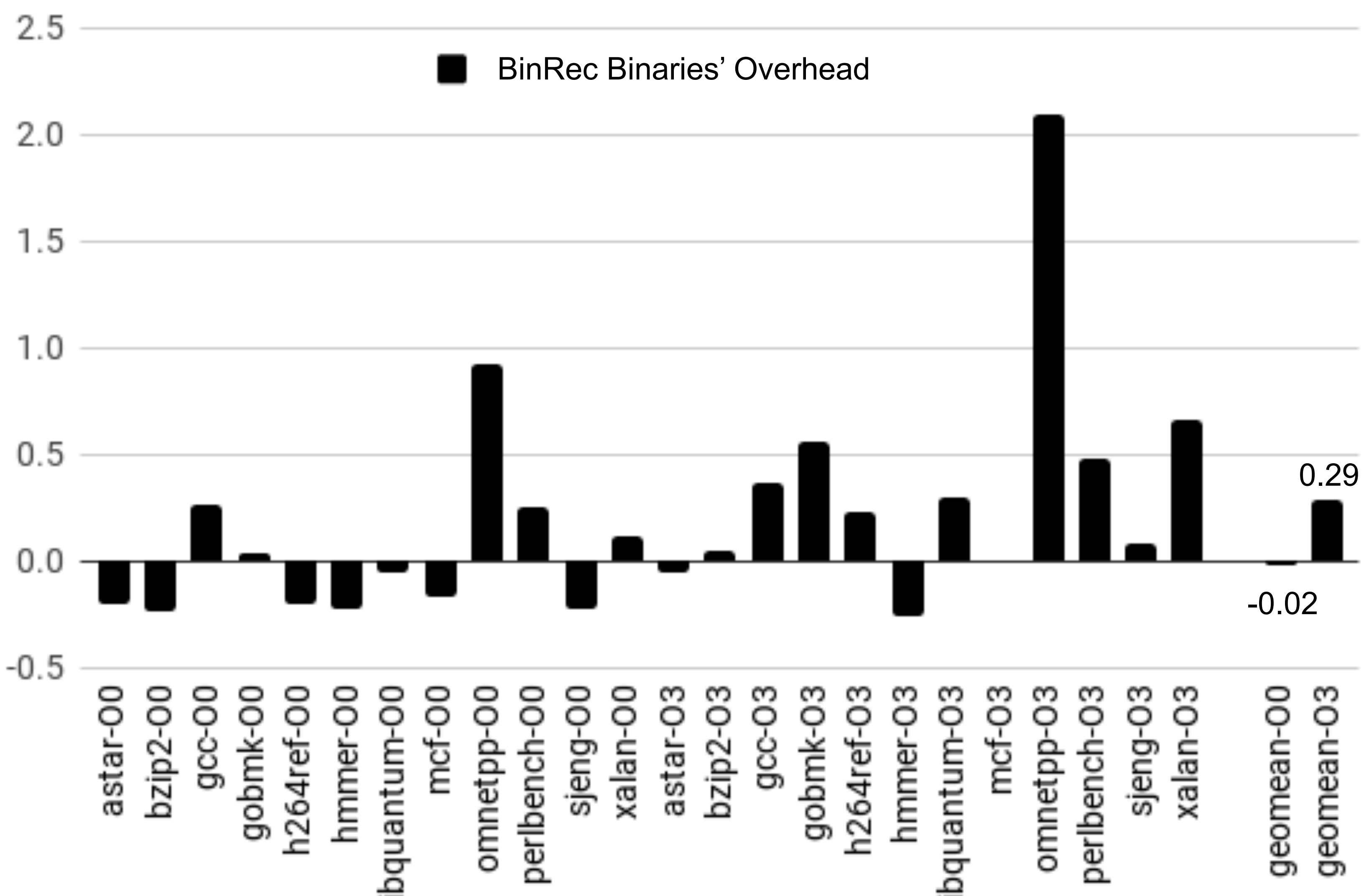
- Use logged 'path misses' as points to restart lifting



Incremental Lifting of Bzip2



Correct and Performant Rewriting of SPEC CINT 2006



BinRec vs Static Rewriters

SPEC Int Geomean	O3
BinRec	1.29x
Multiverse [7]	1.60x
Rev.ng[13]	2.25x

O0	mcf	bzip2	sjeng	libquantum
BinRec	0.83x	0.76x	0.77x	0.95x
McSema	2.31x	2.84x	3.43x	2.07x

- Static approaches are less precise
  - More possible behaviors -> less optimization is possible
- Dynamic lifting has a one-time cost (~450x on SPEC)

SPEC Int Geomean	O0	O3
BinRec	178480s	138379s
McSema	371s	320s

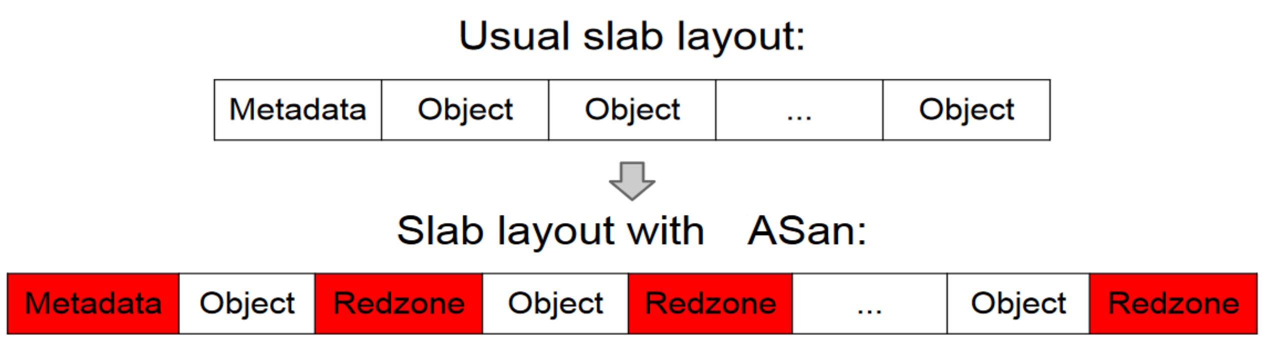
Now we can have nice things!

LLVM IR + dynamic linking support ==

No need to rewrite transformations

Address Sanitizer in BinRec

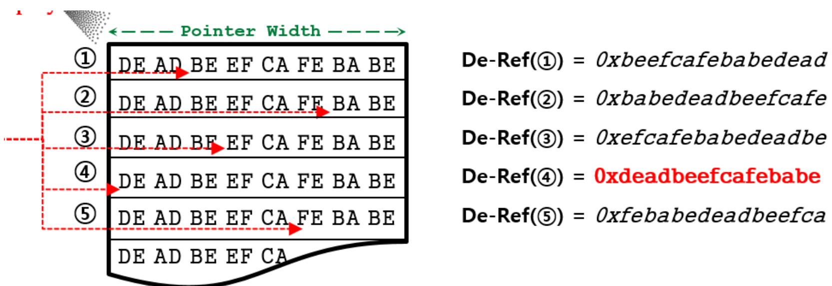
- ASAN: A memory access violation finding tool available in LLVM
- **Works with off the shelf ASAN no modifications on *binaries***
- All memory accesses are instrumented
- Heap allocations are instrumented
- No stack variable symbolization -> stack allocations are not instrumented by ASAN
- ASAN runtime library links and reports violations



[14]

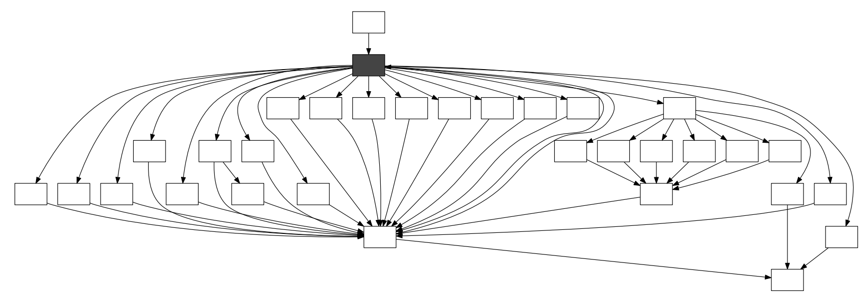
Obfuscation and Ill-formed Code

Unaligned / Overlapping Instructions

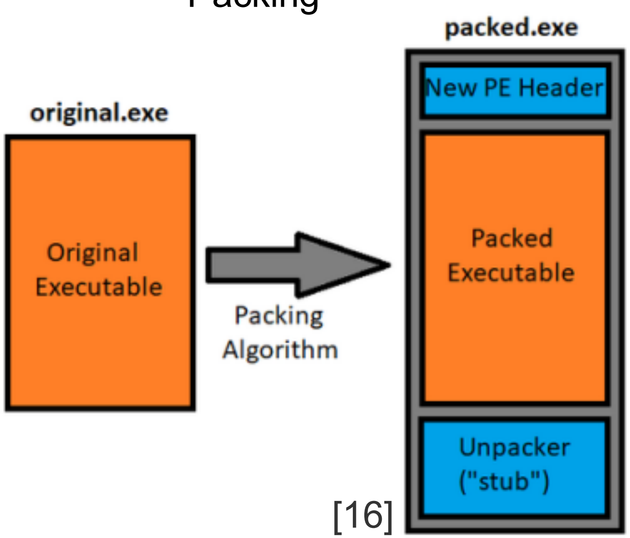


[15]

Virtualization

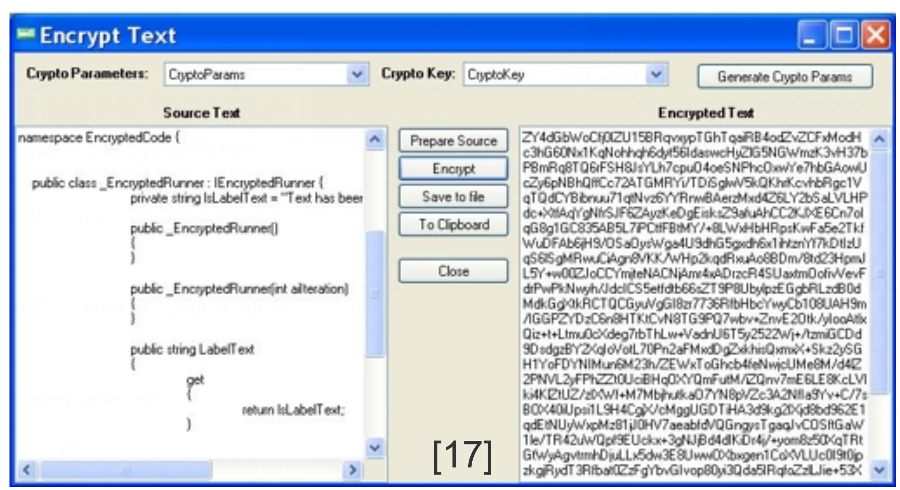


Packing



[16]

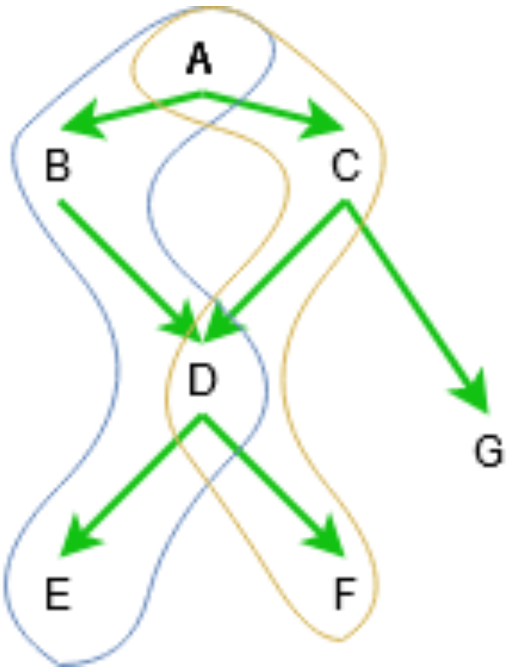
Code Encryption



[17]

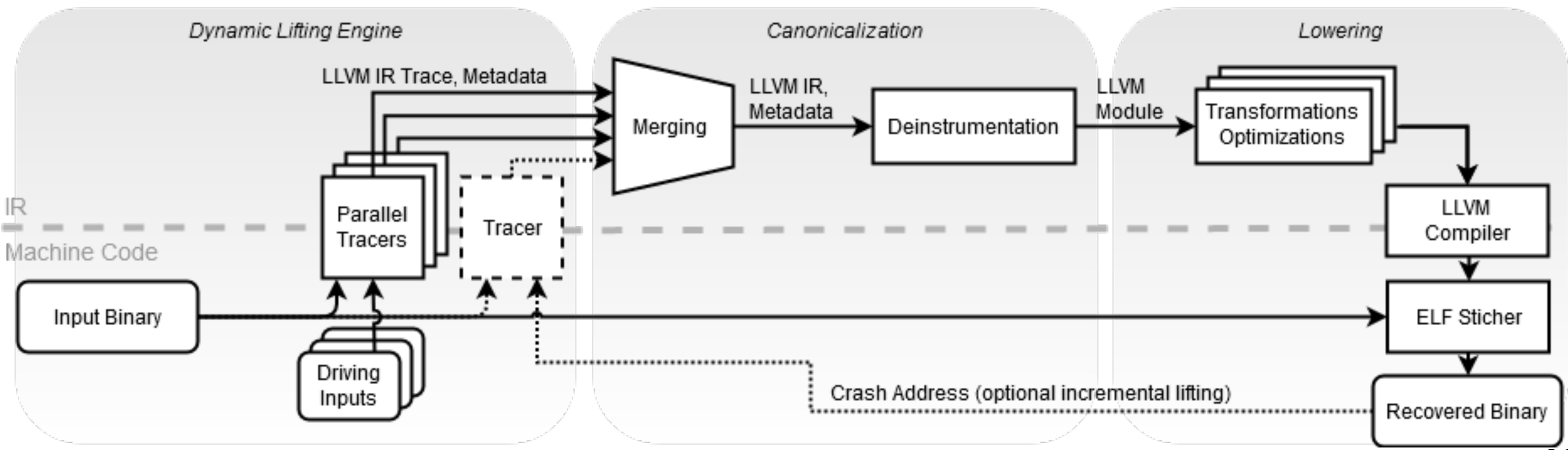
Control-Flow Integrity in BinRec

- Only observed control flows are allowed
  - C -> G disallowed
- Contexts are merged
  - Performance Vs Precision
- Indirect CFT -> Direct CFT
  - Ret = switch %pc, label %error
    [i32 &D, label %BB_D]
- BinCFI uses an address taken heuristic over-approximation
  - BinRec is on average at least 25x more restrictive than BinCFI



BinRec: Dynamic Binary Lifting and Recompilation

- First of its kind dynamic trace lifting and recompilation of stripped binaries
- Heuristic free and supports obfuscated code
- Enables off-the-shelf transformations, which only existed for source code
- Low overhead (29%)



Thanks and Acknowledgements

- We thank our shepherd and the anonymous reviewers for their feedback.
- Thanks to Alyssa Milburn for editing assistance, and Chinmay Deshpande for testing and ongoing efforts.
- This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 andFA8750-15-C-0085, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, by the National Science Foundation under awards CNS-1619211and CNS-1513837. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

Citations

1. "Computer History Museum - 108" by phrenologist is licensed under CC BY-NC 2.0
2. https://arstechnica.com/gadgets/2017/11/microsoft-patches-equation-editor-flaw-without-fixing-the-source-code/
3. R.Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computerprograms. The Computer Journal,1980.
4. https://www.hex-rays.com/products/ida/
5. B. Dolan-Gavitt, T. Leek, J. Hodosh, W. Lee. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. 20th ACM Conference on Computer and Communications Security (CCS), Berlin, Germany, November 2013
6. https://www.trailofbits.com/research-and-development/mcsema/
7. Erick Bauman,Zhiqiang Lin,and Kevin W Hamlen. Supersetdisassembly:Staticallyrewritingx86binarieswithouth euristics. InNDSS, 2018.
8. KapilAnand,MatthewSmithson,KhaledElwazeer,AparnaKotha,Jim Gruen,NathanGiles,andRajeevBarua. Acompiler-levelintermediate representationbasedbinaryanalysisandrewritingsystem. InEurosys, 2013.
9. VitalyChipounov,VolodymyrKuznetsov,andGeorgeCandea. S2E: a platform for in-vivo multi-path analysis of software systems. 2012
10. Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis,RetrofittingsecurityinCOTSsoftwarewithbinaryrewriting, Proc. 26th IFIP TC Int. Information Security Conf. (SEC),2011, pp. 154–172.
11. Mingwei Zhang and R. Sekar,ControlflowintegrityforCOTSbinaries, Proc. 22nd USENIX Security Sym., 2013,pp. 337–352.
12. http://www.ijg.org/
13. AlessandroDiFederico,MathiasPayer,andGiovanniAgosta. Rev.Ng: AunifiedbinaryanalysisframeworktorecoverCFGsandfunction boundaries. In Proceedings of the 26th International Conference on Compiler Construction,CC2017,pages131–141,NewYork,NY,USA, 2017.ACM.
14. Andrey Konovalov, Dmitry Vyukov, LinuxCon 2015 https://events.static.linuxfound.org/sites/events/files/slides/LinuxCon%20North%20America%202015%20KernelAddressSanitizer.pdf
15. Jang, Daehee, et al. "Rethinking Misalignment to Raise the Bar for Heap Pointer Corruption." *arXiv preprint arXiv:1807.01023* (2018).
16. https://kindredsec.com/2020/01/07/the-basics-of-packed-malware-manually-unpacking-upx-executables/
17. http://archeanpartners.com/modules/com_eventlist/?encrypted-code-1449