Enhanced eBPF verification and eBPF-based runtime safety protection

Guang Jin (guang@trustedst.com), Jason Li (jason@trustedst.com) & Greg Briskin (greg@trustedst.com)

Trusted Science & Technology, Inc

Outline

- Introduction of TRUSTEDST
- Background of eBPF
- Formal methods on eBPF verification
- eBPF applications for cyber security

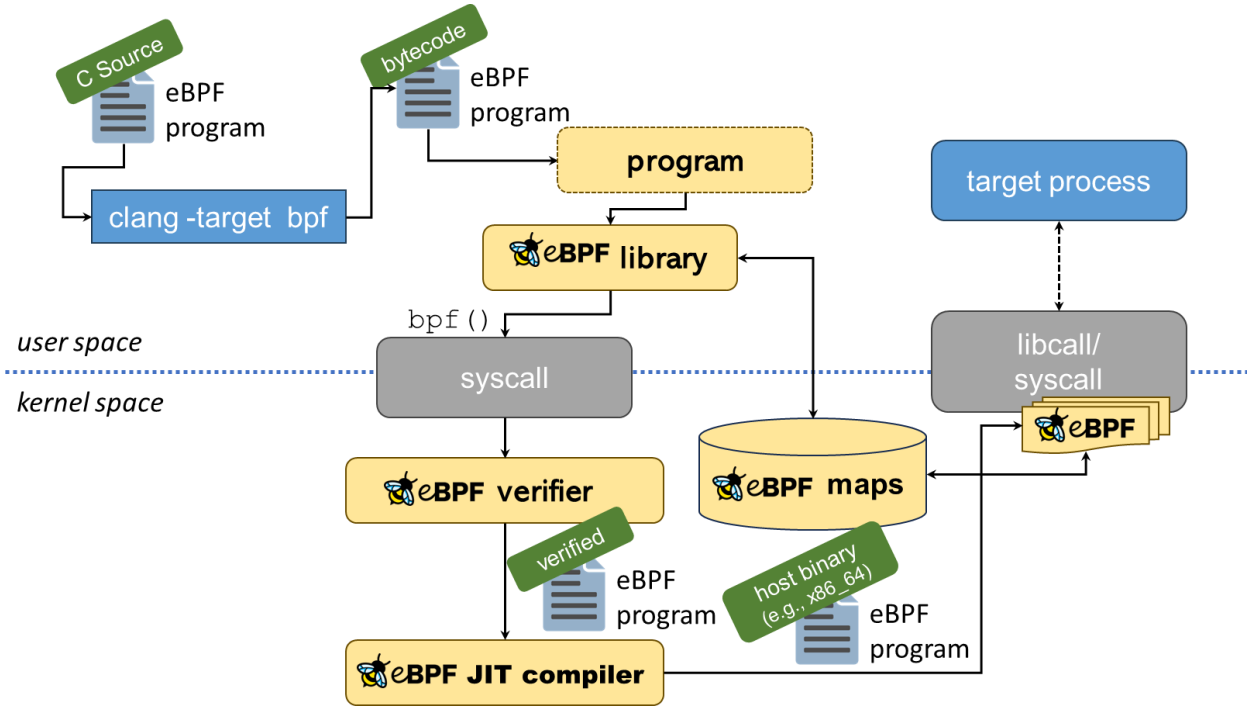
Trusted Science & Technology, Inc.

- Small business entity incorporated at Rockville, Maryland
- Expertise in cyber security, simulation and emulation, embedded systems, and network/communication technologies
- Extensive R&D experience from BAA/SBIRs/STTRs
  - Numerous BAA/SBIR contracts with DoD agencies (e.g., DARPA, AFRL, ARL, and Navy)
  - $18+M DoD S&T contract award during past 5 years
  - Strong influencer in DoD Cyber COI: Cyber security, highly assured system architecture, simulation and emulation, cryptographic solution design/development
- Dynamic R&D focus corporate culture

eBPF background

extended Berkeley Packet Filter (eBPF)

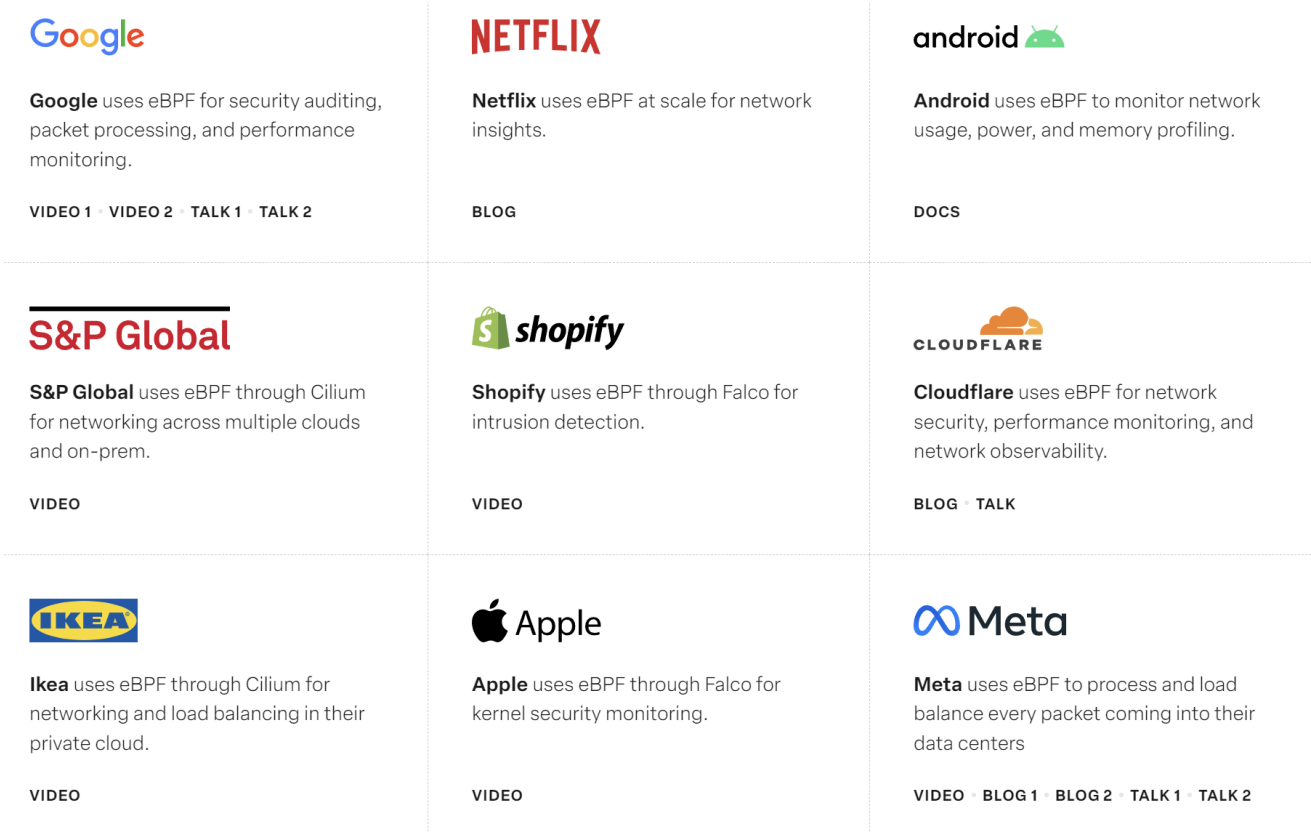
- An *Operating System* (OS) extension to ***safely*** run ***third-party*** code "within the kernel"
  - *Compile Once, Run Everywhere* (CORE), so normal kernel modules are not suitable
- eBPF *Virtual Machine* (VM) in kernel



- 11 64-bit registers: 9 common registers, 1 read-only stack pointer, and an implicit program counter
- supports 100 instructions under 8 classes
- a 512-byte stack
- each eBPF program must be within the 4096 instructions limit, although multiple eBPF programs can be tailed

eBPF Eco-System

- BPF was originally invented for networking (via filtering network packets)
- eBPF introduces new hooking mechanisms
  - suitable for general cyber-security scenarios
- Network-related applications
  - firewall, intrusion detection, etc.
- Container-based cloud management
  - load balancing, security enhancement, etc.

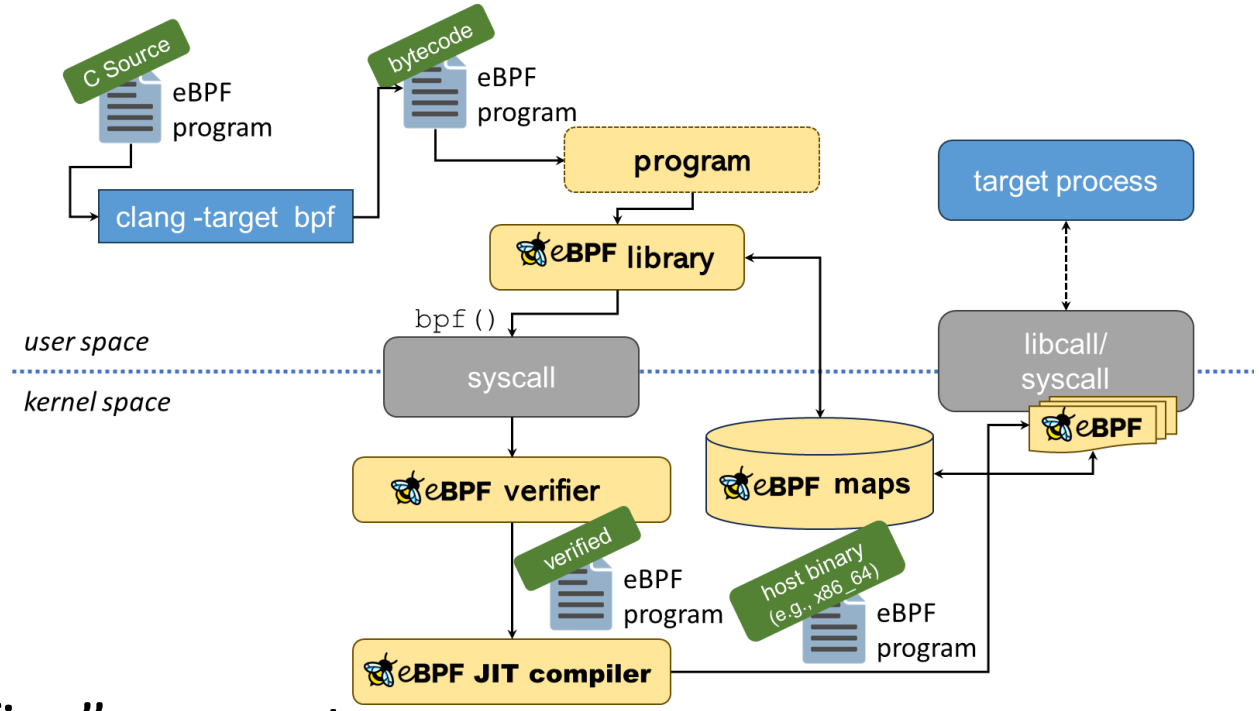


formal methods for eBPF verification

on eBPF programs

Verification Methods in the eBPF Community

- Restricted C/assembly eases the application of formal methods
  - no floating-point numbers, no **unbounded** loops, no direct access to kernel memory/APIs (via helper functions instead), etc.
- Highly constrained memory model
  - single-thread with 512-byte stack, no heap, 4K instruction limit, etc.
- High acceptance of the “loader/verifier” concept
  - the eBPF community is well-aware of the “verifier” concept
  - verifier ensures that the eBPF program is safe to run
  - *Just-in-Time* (JIT) compiler translates eBPF bytecodes into the host machine instructions



Formal Verification of eBPF Programs

Abstract Interpretation

- Linux's in-kernel ebpf verifier: static analysis based on (unsound) abstract interpretation
  - code: kernel/bpf/verifier.c
  - unsoundness: it has been shown that Linux's verifier may falsely pass unsafe eBPF programs
- PREVAIL: abstract interpretation on eBPF programs
  - code: https://github.com/vbpf/ebpf-verifier
- Pro
  - verification is usually fast (different domains may increase computation costs)
  - false negatives are impossible under sound analysis
- Cons
  - false positives are unavoidable due to the over-approximation

Symbolic Execution

- radius2: fast symbolic execution and taint analysis covering different architectures (including eBPF bytecodes)
  - code: https://github.com/aemmitt-ns/radius
- angr: a binary analysis and symbolic execution framework also supports eBPF
  - code: https://github.com/angr/angr-platforms/tree/master/angr_platforms/bpf
- ExoBPF: proof-carrying with symbolic execution
  - code: https://github.com/uw-unsat/exoverifier
- Pros
  - security proven with synthesized exploits
  - some proofs can be automatically generated into 1st order logics
- Cons
  - verification is slow
  - some cases cannot be solved automatically
  - loops cause non-termination

Example #1

```
1: r1 = r2
2: if r1 >= 0x8 goto out
3: r0 += r2
4: *(u8 *)(r0) = 0

5: out: exit
```

non-relational interval domain:
$r1 \in [0, U_{64}MAX], r2 \in [0, U_{64}MAX]$

symbolically execute this path only
$r1 = r2 \in [0, 7]$

$r1 \in [0, 7], r2 \in [0, U_{64}MAX]$

check if the index 7 is within the array bound

assume $r0$ points to an array

an out-of-bound access is falsely identified

- Linux eBPF verifier used to falsely reject this program
  - non-relational domains cause this type of false-positives
- Abstract interpretation (even over sound relational domains) based analysis cannot completely remove false positives
- Symbolically execute this (potentially unsafe) path with the help of an SMT solver
  - embed condition formulas and solve formulas

Example #2

```
<LBB0_1>:
1: r2 = r10
2: r2 += -0x28
; cumul +=array[index]
3: r0 += r2
;index++;
4: r1 += 0x1
;while ( index< 40 ) {
5: if r1 != 0x28 goto LBB0_1
```

invariants:
$r1 \in [0, U_{64}MAX], r1 \neq 40$ and $LC@5 \in [0, +\infty]$

introduce $LC@5$ as a concrete state

a ghost variable $LC@5$ can be introduced and paired with $r1$

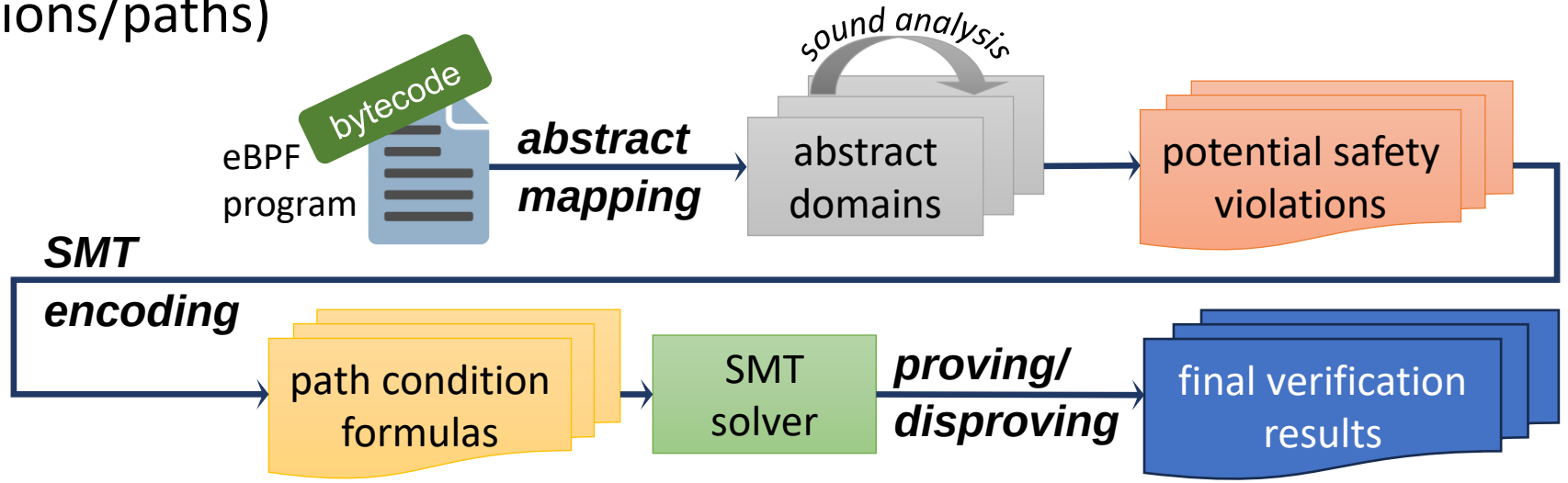
an infinite loop is falsely identified

symbolically execute this loop, update $LC@5 \ += 1$ and check if $LC@5$ exceeds the bound limit

- Since Linux v5.3, loops are allowed in eBPF
  - to prove the reachability, a symbolic execution may get trapped within loops
- PREVAIL, based on sound analysis, **falsely rejects** this program
  - as invariants are over-approximated, false positives are unavoidable
  - ghost variables are expensive and hard to deal with
- Symbolically execute the loops, but check the concrete states against security specifications

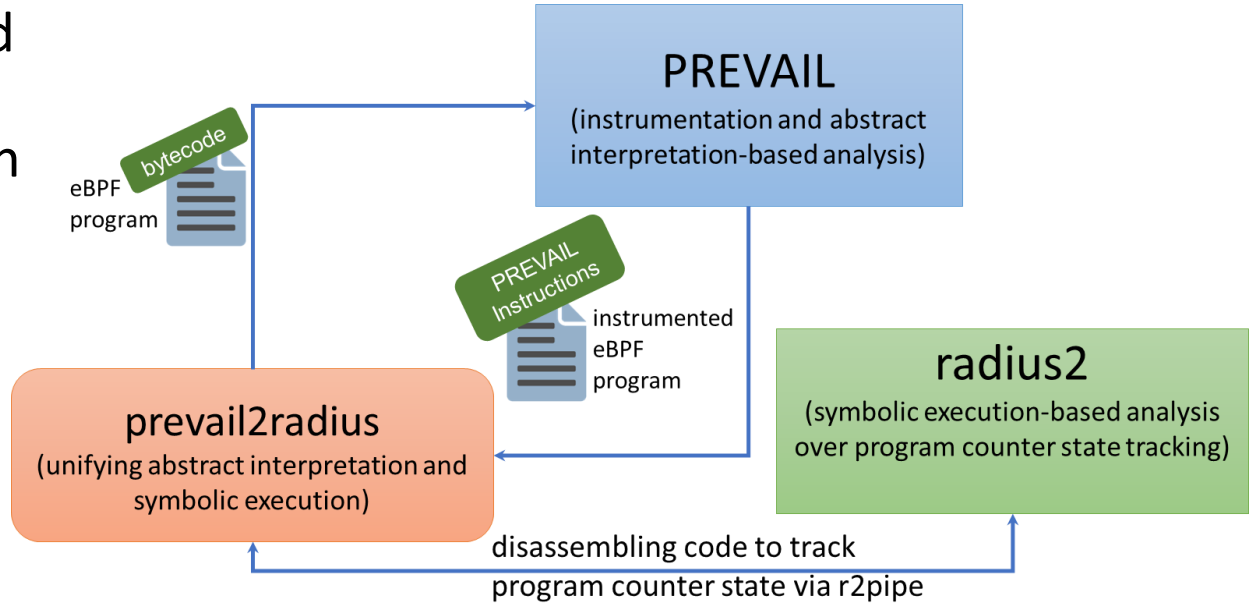
Our Approach

- Combination of both abstract interpretation and symbolic execution
- Steps
  - 1. use sound abstract interpretation to locate safety violations (i.e., identify insecure functions/paths)
  - 2. ignore safe paths, only convert the insecure paths into condition formulas
  - 3. use an automatic SMT solver to prove the validity formulas (e.g., removing false positives and providing violation conditions)

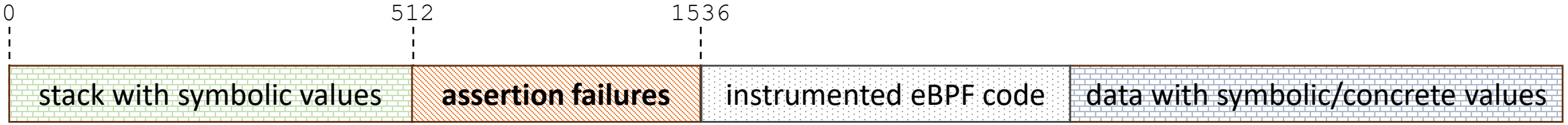


Our Prototype

- Our prototype is based on both PREVAIL and radius2
- Via PREVAIL, eBPF code is instrumented with security checks as Crab *Intermediate Representation* (IR)
- radius2 uses Boolector as the SMT solver to explore *Program Counter* (PC) states
- **prevail2radius** is our mediator to unify abstract interpretation and symbolic execution
  - prevail2radius reuses PREVAIL's eBPF disassembler and instrumentation
  - prevail2radius translates Crab IR into *Evaluable Strings Intermediate Language* (ESIL) for radius2 to track PC states
  - prevail2radius introduces new PC states to represent the assertion failures
  - prevail2radius uses additional virtual registers to encode the path conditions for instrumented assertions



Memory Layouts of eBPF Programs under Verification



- Memory mapping
  - 0-511: stack; 512-1535: assertion failures; remaining addresses are used by eBPF instructions/assertions and data segment (e.g., BTF structures pointed by `ctx`, loop counters, etc.)
- A single assertion is often translated into multiple ESIL instructions
  - ✓ multiple conditional branches and `mov`s
  - ✓ e.g., a stack access is represented by four instructions for lower and upper bound checks
- PC states to be explored by the symbolic execution
  - ✓ Solving the path conditions with the SMT solver

Initial Results

- Current prototype can verify some eBPF programs
- Crab IRs are disassembled into ESILs
- Assertions are translated into conditional branches
  - If a failure is triggered, PC goes to the lower 1K space
- *ctxoffset.o* has an integer overflow at Line 3
- The prototype can use symbolic execution to find the concrete inputs to trigger the integer overflow

```
ebpf@ebpfx86:~/radare/radius/target/debug$ ./radius2 -p prevail:/home/ebpf/prism
.tmp.bpf -s x 64n -S r1 x 64
0x61c6af338b50      r7 = r1                                  |  r1,r7,=
0x61c6af338b58      r1 = 0                                   |  0,r1,=
0x61c6af338b60      r11 = r10                                |  r10,r11,=
0x61c6af338b61      r11 -= 1                                 |  1,r11,-=
0x61c6af338b62      assert((r10 + -1) >= 0)                  |  0,r11,>=,?{,0x61
c6af338b63,}{,0x0,},pc,=
0x61c6af338b63      r11 += 1                                 |  1,r11,+=
0x61c6af338b64      assert((r10 - 1) <= eBPF_stack_size)     |  512,r11,<=,?{,0x
61c6af338b68,}{,0x1,},pc,=
0x61c6af338b68      *(u8 *)(r10 + -1) = r1                   |  r1,-1,r10,+,=[8]
```

```
<func>:
1: r2 = 0x1
2: *(u32 *)(r10 - 0x4) = r2
3: r1 += 0x8
4: r3 = r10
5: r3 += -0x4
6: r2 = 0x0 ll
7: r4 = 0x0
8: call 0x35
9: exit
```

```
symbolic PC : (ite (= a #xfffffffffffffff8) #x0000000000000007 #x00005c5441ee011
0)
0x5c5441ee0110      ignore_assert(r2.type == map_fd)         |  r2,r2,=
0x00000007      context offset zero failure              |  7,TRAP
crash found
0x5c5441ee0118      ignore_assert(r3.type in {stack, packet}) |  r3,r3,=
0x5c5441ee0120      ignore_assert(within stack(r3:key_size(r2))) |  r11,r11,=
0x5c5441ee0128      ignore_assert(r4.type == number)         |  r4,r4,=
0x5c5441ee0130      call 53                                  |  pc,sp,=[8],8,sp,
-=,0x35,$
0x5c5441ee0138      ignore_assert(r0.type == number)         |  r0,r0,=
0x5c5441ee0140      exit                                     |  8,sp,+=,sp,[8],p
c,=
state.backtrace.is_empty():true new_flags.is_empty():true
  a : 0xfffffffffffffff8
```

Initial Results (cont.)

```
Post-invariant: [
    meta_offset=[-4098, 0],
    packet_size=[0, 65534],
    pc[7]=[1, +oo],
    r0.type=uninitialized,
    r1.svalue=40, r1.type=number, r1.uvalue=40,
    r10.stack_offset=512, r10.svalue=[512, 2147418
    s[472...479].svalue=0, s[472...479].uvalue=0,
value=0, s[496...503].svalue=0, s[496...503].uvalu
Stack: Numbers -> {[472...511]}

7: Lower bound must be at least 0 (valid_access(r2
7: Upper bound must be at most EBPF_STACK_SIZE (va
7: Stack content is not numeric (valid_access(r2.o
7:  (r0.type in {number, ctx, stack, packet, share
7:  (r2.type in {number, ctx, stack, packet, share
7: Only numbers can be added to pointers (r2.type
13:14:  (r0.type == number)

Could not prove termination.
Program terminates within 2147483647 loop iteratio
0,0.196174,6656
```

```
x599299c1d678   r11 = *(u16 *)(r15 + 27)                    |  27,r15,+,[4],r11,=
x599299c1d679   r11 += 1                                    |  1,r11,+=
x599299c1d67a   assert(loop@27 <= 4095)                     |  4095,r11,<=,?{,0x599299c1d67b,}{,
x599299c1d67b   *(u16 *)(r15 + 27) = r11                    |  r11,27,r15,+,=[16]
x599299c1d680   if (r1 != 40) goto 0x599299c1d5e0, else goto 0x599299c1d688 |  40,r1,-,?{,0x
x599299c1d5e0   r2 = r10                                    |  r10,r2,=
x599299c1d5e8   ignore_assert(r2.type in {number, ctx, stack, packet, shared}) |  r2,r2,=
x599299c1d5f0   r2 += -40                                   |  -40,r2,+=
x599299c1d5f8   ignore_assert(r2.type in {number, ctx, stack, packet, shared}) |  r2,r2,=
x599299c1d600   ignore_assert(r1.type in {number, ctx, stack, packet, shared}) |  r1,r1,=
x599299c1d608   ignore_assert(r1.type in {ctx, stack, packet, shared} -> r2.type == number)
x599299c1d610   ignore_assert(r2.type in {ctx, stack, packet, shared} -> r1.type == number)
x599299c1d618   r2 += r1                                    |  r1,r2,+=
x599299c1d620   ignore_assert(r2.type in {ctx, stack, packet, shared}) |  r2,r2,=
x599299c1d628   ignore_assert(valid_access(r2.offset, width=1) for read) |  r2,r2,=
x599299c1d630   r2 = *(u8 *)(r2 + 0)                        |  0,r2,+,[1],r2,=
x599299c1d638   ignore_assert(r0.type in {number, ctx, stack, packet, shared}) |  r0,r0,=
x599299c1d640   ignore_assert(r2.type in {number, ctx, stack, packet, shared}) |  r2,r2,=
x599299c1d648   ignore_assert(r2.type in {ctx, stack, packet, shared} -> r0.type == number)
x599299c1d650   ignore_assert(r0.type in {ctx, stack, packet, shared} -> r2.type == number)
x599299c1d658   r0 += r2                                    |  r2,r0,+=
x599299c1d660   ignore_assert(r1.type in {number, ctx, stack, packet, shared}) |  r1,r1,=
x599299c1d668   r1 += 1                                     |  1,r1,+=
x599299c1d670   ignore_assert(r1.type == number)            |  r1,r1,=
x599299c1d678   r11 = *(u16 *)(r15 + 27)                    |  27,r15,+,[4],r11,=
x599299c1d679   r11 += 1                                    |  1,r11,+=
x599299c1d67a   assert(loop@27 <= 4095)                     |  4095,r11,<=,?{,0x599299c1d67b,}{,
x599299c1d67b   *(u16 *)(r15 + 27) = r11                    |  r11,27,r15,+,=[16]
x599299c1d680   if (r1 != 40) goto 0x599299c1d5e0, else goto 0x599299c1d688 |  40,r1,-,?{,0x
x599299c1d688   ignore_assert(r0.type == number)            |  r0,r0,=
x599299c1d690   exit                                        |  8,sp,+=,sp,[8],pc,=
o crash dected
```

- Loop counters are tracked as concrete states by the symbolic execution
- Our current prototype can avoid false positives

More Features

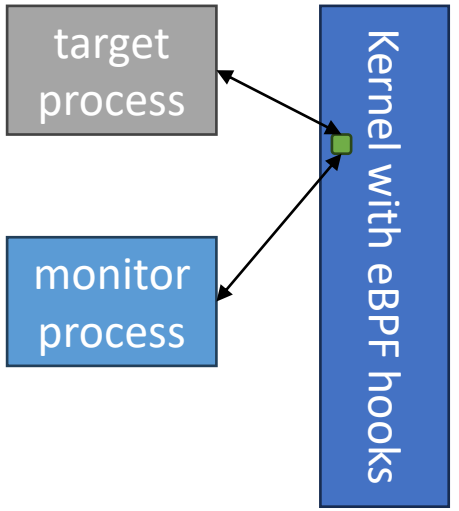
- Automate the symbolic/concrete state assignment by parsing the *BPF Type Format* (BTF) data
  - minimize human involvement
- Support new eBPF standards and features
  - function calls, loops, etc.
- Minimized false positives
  - via formal methods
- Automatic violation condition generation
  - synthesize exploits
- Direct and intuitive feedbacks to eBPF programmers

eBPF-based security solutions

and our approach

Runtime Safety Protection

- eBPF enables dynamic instrumentation via various hooking mechanisms
  - e.g., `kprobe`, `kretprob`, `uprobe` and `uretprobe`
  - on almost any kernel/user-space routines (but not on in-lines or function-like macros)
- Dynamically and non-disruptively monitor and profile binary executables
  - without offline instrumentation or rewritting (on binary or source code)
- Dynamic patching is possible
  - via revised *Call Flow Graph* (CFG) based on eBPF hooks
- Context switching dominates the runtime overheads
  - typically, two context switches are required
  - user-space eBPF hooking (e.g., uBPF, rbpf and bpftime) can reduce the overheads



Our Approach

- Apply eBPF hooking mechanisms to enable an extensible runtime security monitoring
- Target
  - user-space natively-compiled binary executables
- Methods
  - implement eBPF programs hooking on syscalls/libcalls
  - build various analysis plugins for different security/performance issues
  - develop intuitive web-based graphical interface for human analysts
  - interface AI/ML for advanced analysis

Example #1: heap analyzer

```
SEC("uprobe/libc.so.6:malloc")
int malloc_hook(struct pt_regs *ctx){
                    ⋮
SEC("uprobe/libc.so.6:free")
int free_hook(struct pt_regs *ctx){
                    ⋮
```

hooks are implemented as eBPF programs

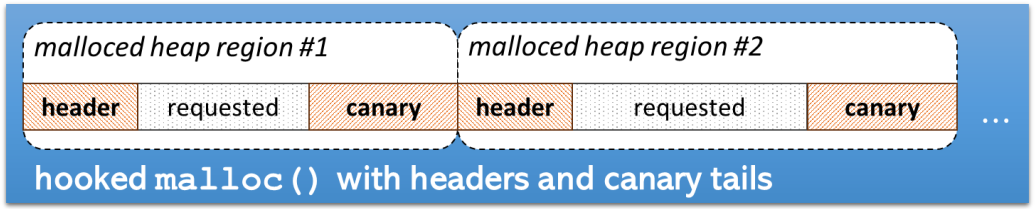
profiler process does further analysis

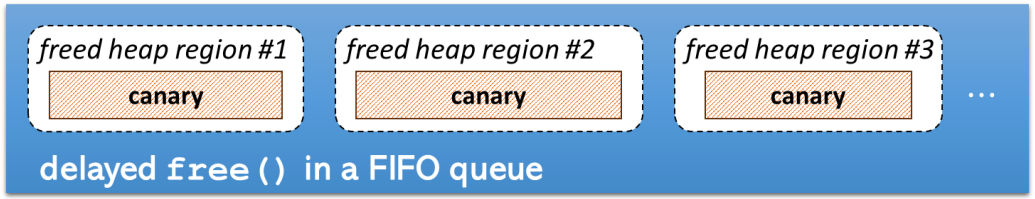
```
        pid=4558        malloc calls: 219 free calls: 239
20:48:55
        pid=4558        malloc calls: 222 free calls: 242
20:48:56
        pid=4558        malloc calls: 226 free calls: 246
:48:57
        pid=4558        malloc calls: 230 free calls: 250
:48:58
        pid=4558        malloc calls: 233 free calls: 253
20:48:59
        pid=4558        malloc calls: 241 free calls: 261
20:49:00
        pid=4558        malloc calls: 245 free calls: 265
20:49:01
        pid=4558        malloc calls: 249 free calls: 269
20:49:02
        pid=4558        malloc calls: 257 free calls: 277
20:49:03
```

- eBPF hooks are attached at `malloc` and `free`
- Potential security applications
  - memory profiling, heap usage
  - detection of double-free errors (CWE-415)
  - detection of multiple releases of same resource (CWE-1341) by applying similar hooks on other resource allocations and deallocations (e.g., `open` and `close`)

Example #1: heap analyzer (cont.)

- Same eBPF hooks at `malloc` and `free`
- More tracking data added to heaps regions
  - headers and canary tails
- Detection of more memory-corruption errors
  - heap leaks (CWE-244)
  - use-after-frees (CWE-416)
  - heap overflows (CWE-122)
- Hooks are dynamically attached to target processes
  - **no offline instrumentation**





Example #2: mutex analyzer

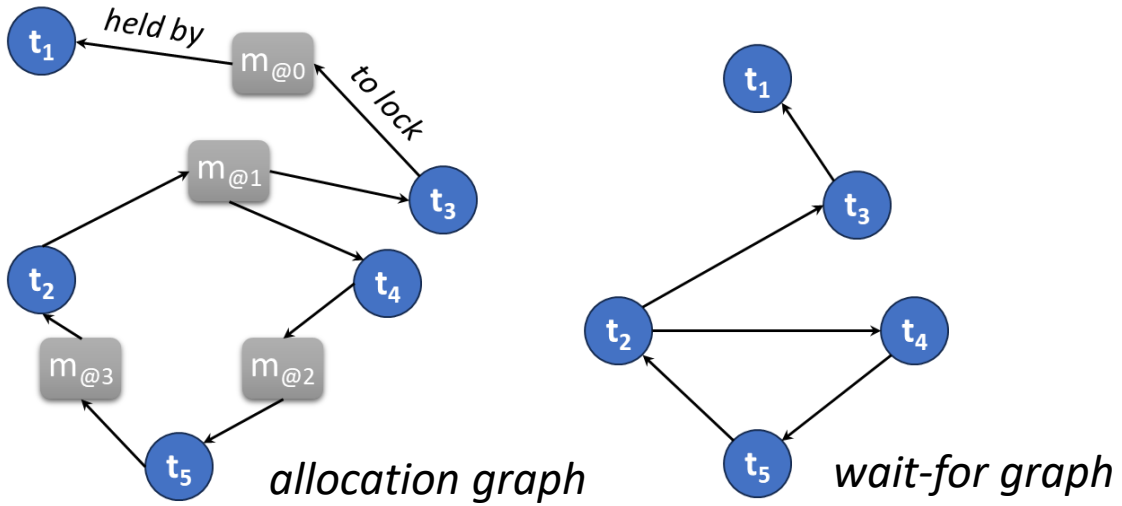
```
SEC("uprobe/libc.so.6:mtx_init")
int mtx_init_hook(struct pt_regs *ctx){
                        ⋮
SEC("uprobe/libc.so.6:mtx_lock")
int mtx_lock_hook(struct pt_regs *ctx){
                        ⋮
SEC("uprobe/libc.so.6:mtx_unlock")
int mtx_unlock_hook(struct pt_regs *ctx){
                        ⋮
```

hooks are implemented as eBPF programs

- eBPF hooks are attached with mutex-related `libc` functions
- Potential security applications
  - general lock profiling
  - detection of dead locks (CWE-833)
  - detection of unlock of a resource that is not locked (CWE-1341)

Example #2: mutex analyzer (cont.)

- Same eBPF hooks help to construct allocation and wait-for graphs
- Apply similar hooks on other resources to detect more security issues
  - on spin-locks, files, etc.
  - detection of dead locks (CWE-833)
    - loop detection in wait-for graphs
  - resource starvation errors (CWE-400)
  - live lock issues (CWE-667)



Example #3: context-aware access control

```
SEC("uprobe/libc.so.6:open")
int open_hook(struct pt_regs *ctx){
                        ⋮
SEC("uprobe/libc.so.6:close")
int close_hook(struct pt_regs *ctx){
                        ⋮
SEC("uprobe/libc.so.6:socket")
int socket_hook(struct pt_regs *ctx){
                        ⋮
```

hooks are implemented as eBPF programs

- eBPF hooks are attached with file/network-related `libc` functions
- Potential security applications
  - information flow tracking
  - dynamic context-aware multi-level access control
  - fine-grained dynamic firewall

Take-aways and Expectations

- eBPF's constrained environment is ideal for formal methods
  - the security of eBPF programs can be formally proven
- With the eBPF development, more security centric functionalities are being added
- TRUSTEDST is improving the eBPF verifier
  - supporting new eBPF features
  - integrating more formal methods
- TRUSTEDST is building more effective and efficient eBPF based security solutions
  - building a unified framework for various security applications
  - building an extensible framework with different eBPF programs for security analysis and monitoring
  - compatible with prebuilt binary executables
  - no need of offline instrumentation or rewriting