실행 중인 Roblox 클라이언트에 FastFlag 를 기록하기: 읽기 우선 주입 경로와 결과 기반 검증

bak · 오픈소스 기여자들

Pandora 프로젝트

문서 PANDORA-TECH-002 · **판** v1.1.0 (2026-09) · **대상** Windows 10/11 x64
측정 단일 기기, 벽시계, 본문에 적은 판. 재지 않은 것은 재지 않았다고 적는다.

초록

Roblox 의 FastFlag 는 클라이언트 동작을 바꾸는 이름–값 항목이다. 통용되는 방법은 클라이언트가 켜질 때 읽는 시작 시점 파일(`ClientAppSettings.json`)에 목록을 써 두는 것이다. 현재 클라이언트는 이 파일에서 온 항목을 전부 받아들이지 않는다. 허용되지 않은 이름은 조용히 버려지고 클라이언트 로그에 `Denied local configuration for: <이름>` 한 줄만 남는다. 즉 이 경로로 실제 적용되는 것은 제한된 부분집합이며, 같은 파일을 쓰는 다른 도구도 같은 제한을 받는다. 또한 이 방법은 실행 중 변경이 불가능하고, 어떤 항목이 반영됐는지 프로그램이 확인할 수단을 주지 않는다. 이 논문은 실행 중인 클라이언트의 플래그 레지스트리를 메모리에서 찾아 값 슬롯에 직접 쓰는 경로와, 그 경로가 통과해야 하는 검사를 기술한다. 설계 전제는 두 가지다. 모든 쓰기 앞에는 그 쓰기를 정당화하는 읽기가 있어야 하고, 성공은 쓴 횟수가 아니라 게임이 현재 들고 있는 값으로 센다. 구조 판정은 단일 시그니처 대조가 아니라 독립 증거의 누적으로 하며, 증거가 모자라면 아무것도 쓰지 않는다. 302개 표에 대한 읽기 전용 감사에서 이 빌드가 가진 이름 75개 중 75개가 요청한 값을 들고 있었고, 값이 다른 항목과 값 슬롯을 찾지 못한 항목은 각각 0개였다. 넣지 못한 227개가 이 빌드에 없는 이름이라는 판정은 모듈 문자열표와 공개 설정표로 따로 확인했다. 인덱스에서 플래그가 아닌 항목 3,300개를 제거하자 주입 가능 비율은 97.0%에서 표본 10,242개 중 10,240개로 올라갔고, 실행 후 첫 플래그가 들어가기까지의 지연은 약 8초에서 약 1.0초로 줄었으며 같은 구간에서 적용 결과는 715개 중 435개로 변하지 않았다.

키워드 FastFlag, 프로세스 메모리 쓰기, 타입 계약, 증거 기반 구조 판정, 감사

1. 서론

FastFlag 는 Roblox 클라이언트가 실행 중 참조하는 설정 항목이며, 렌더링·스케줄러·디버그 경로 등 내부 동작을 바꾼다. 사용자가 이 값을 바꾸는 통상적인 경로는 클라이언트 실행 전에 설정 파일 (`ClientAppSettings.json`)에 목록을 써 두는 것이다.

이 경로의 제약은 세 가지이고, 첫째가 가장 크다. **클라이언트는 이 파일의 항목을 전부 받아들이지 않는다.** 허용 목록 밖의 이름은 적용되지 않고, 그 사실은 클라이언트 로그의 `Denied local configuration for: <이름>` 한 줄로만 남는다(2.3절). 둘째, 값을 바꾸려면 클라이언트를 다시 켜야 한다. 셋째, 적은 이름이 그 빌드에 실제로 존재했는지, 값이 반영됐는지 확인할 방법이 없다. 목록의 오타나 바뀐 이름은 조용히 무시된다.

이 논문은 다른 경로를 기술한다. 실행 중인 프로세스의 플래그 레지스트리를 메모리에서 찾아 값 슬롯에 직접 쓰고, 쓴 결과를 다시 읽어 확인하며, 넣지 못한 항목에 대해서는 이름 단위로 사유를 남긴다. 구현은 Windows x64 단일 실행 파일(Pandora)이며[3], 주입 경로는 약 2,700줄의 엔진과 약 17,000줄의 실행 경로로 나뉜다. 엔진은 프로세스 핸들에 의존하지 않아 Roblox 없이 단위 시험이 가능하고, 현재 단위 시험은 160개다.

기여는 다음과 같다.

1. 읽기 우선 주입 경로. 프로세스 메모리 접근부터 감사 기록까지 여섯 계층으로 나누고, 어떤 경로도 그 쓰기를 정당화하는 읽기를 거치지 않고 쓰기에 도달하지 못하게 했다(8절).
2. 증거 기반 구조 판정. 고정 오프셋 대신 서로 독립인 증거를 모아 레지스트리를 판정하고, 판정에 쓰는 배치 상수 자체를 실행 중 표본으로 배운다(6절, 7.3절).
3. 결과 기반 검증. 쓰기 횟수가 아니라 메모리를 다시 읽어 세는 읽기 전용 감사와, "이 빌드에 없다"는 판정을 구현 바깥의 증거로 확인하는 절차(10.2, 10.3절).
4. 지연 분해. 실행 후 첫 플래그까지를 네 단계로 나눠 재고 각 단계의 원인을 제거해 약 8초를 약 1.0초로 줄이면서, 적용 결과가 변하지 않음을 이름 집합 수준에서 확인했다(10.5, 10.6절).

2. 배경

2.1 클라이언트 안의 표현

플래그는 이름과 값을 가진 서술자(descriptor) 객체로 존재하고, 이름으로 찾아가는 레지스트리에 등록된다. 레지스트리는 MSVC 구현의 연결 리스트와 해시 맵이며, 키는 접두사를 뗀 이름이다. `FFlag`, `DFFlag`, `FInt`, `DFInt`, `FString`, `FLog` 등의 접두사는 값의 종류와 갱신 경로를 구분할 뿐 키의 일부가 아니다. 따라서 접두사가 다른 두 이름이 같은 슬롯을 가리킬 수 있다.

관측된 배치는 부록 B 에 있다. 같은 경로를 요약한 이전 문서가 있다[2]. 값은 서술자 기준 `+0xC0`, 이름은 `+0xD0`에 (모듈 내 문자열 포인터, 길이) 짝으로 놓인다. 문자열 값은 MSVC `std::string` 레이아웃(`+0x00` 16바이트 union, `+0x10` size, `+0x18` capacity)을 따르며, capacity 가 15 이하이면 버퍼는 객체 안에 있다(SSO).

2.2 실행 중 쓰기가 실패하는 방식

개발 중 관측한 크래시는 세 가지이고, 셋 다 대상 메모리를 읽지 않고 쓴 경우다.

- 낡은 RVA 표를 믿고 모듈베이스+오프셋에 4바이트를 썼다. 그 자리는 살아 있는 포인터였다.
- `std::string` 객체 위에 원시 바이트를 썼다. 다음 접근에서 예외가 발생했다.
- 1바이트 bool 슬롯에 4바이트 int 를 썼다. 이웃한 값 세 개가 함께 손상됐다.

세 경우 모두 쓰기 직전에 한 번 읽었으면 막혔다. 이 관측이 설계의 전제다.

별도의 실패 유형이 하나 더 있다. 클라이언트가 서버를 옮기면 DataModel 을 새로 짓고, 그때 들고 있던 주소는 다른 객체의 것이 된다. 수집한 크래시 로그 4건은 모두 `submitStartGameTask` 직후의 `HardAssert (Could not find class descriptor)` 였다.

2.3 시작 시점 파일 채널과 그 제한

시작 시점 파일은 설치된 클라이언트 버전 폴더의 `ClientSettings/ClientAppSettings.json` 이다. 부트스트래퍼마다 자기 `Versions` 폴더를 두므로 실제 경로는 여럿이다. 클라이언트가 그 파일을 읽었다는 사실은 로그의 `LoadClientSettingsFromLocal` 항목으로 확인된다.

읽은 것과 받아들인 것은 다르다. 클라이언트는 허용 목록 밖의 이름에 대해 `Denied local configuration for: <이름>` 을 남기고 그 값을 쓰지 않는다. 우리는 배치한 키 집합과 그 로그를 대조해 통과·거부를 센다(`--fflag-file-status`). 결과는 `fflags/file-channel.txt` 로 남는다. 거부는 클라이언트 쪽 정책이므로 같은 파일을 쓰는 다른 도구에서도 같다.

파서 자체의 제약도 있다. 접두사 없는 이름(`RenderShadowmapBias` 처럼)은 파일 파서가 그대로 무시한다. 그래서 이 채널을 쓸 때는 인덱스로 이름을 정규화한 뒤 배치한다. 정규화가 곧 적용률이다.

그럼에도 이 채널을 버리지 않는다. 초기화 시점에 한 번만 읽히는 플래그는 프로세스가 뜬 뒤의 쓰기로는 효과가 나지 않으므로, 그런 항목은 이 경로로만 들어간다. 두 채널은 배타적이지 않고 역할이 다르다. 되돌리기도 두 경로를 함께 다룬다(9.1절).

이 빌드에서 어떤 이름이 통과하는지의 전수 조사는 하지 않았다. 도구는 그 순간 배치한 집합에 대해서만 통과·거부를 보고한다.

2.4 설계 원칙

1. **읽기 우선.** 쓰기는 그 자리가 기대한 타입임을 반증하지 못했을 때만 발행한다.
2. **모르면 쓰지 않는다.** 노드의 기본 상태는 "구조를 모름 + 쓰기 금지"이고, 증거가 이 상태를 끌어올린다.
3. **증거는 독립이어야 한다.** 구조적 증거(포인터 모양, 마스크 꼴)만으로는 어떤 객체든 통과한다. 이 맵이 플래그 레지스트리임을 말하는 것은 이름과 타입이다.
4. **성공은 결과로 센다.** 쓴 횟수가 아니라 다시 읽은 값으로 센다. 감사 경로는 쓰기 경로와 분리한다.
5. **거부에는 사유가 붙는다.** 거부 사유는 감사의 1차 자료이고, 사용자 화면의 문구는 그 자료의 요약이어야 한다.

3. 역공학 절차

관측은 도구로 굳힌다. 한 번 본 것은 명령으로 남겨야 다음 빌드에서 같은 관측을 재현할 수 있고, 숫자가 남아야 고친 것이 실제로 나아졌는지 말할 수 있다. 이 절은 앞 절의 배치가 어떻게 나왔는지를 순서대로 적는다. 명령은 부록 A 에 있다.

3.1 이름 문자열에서 노드까지

시작점은 주소가 아니라 이름이다. 알려진 플래그 이름 하나를 힙에서 찾고, 그 문자열을 가리키는 `std::string` 헤더를 찾고, 그 헤더를 품은 구조체가 서술자 포인터를 들고 있는지 본다 (`--fflag-locate`). 여기서 노드의 링크·이름·서술자 오프셋 후보가 나온다.

이 단계에서 서버 설정표와 레지스트리를 반드시 갈라야 한다. 서버 설정표는 서버가 보낸 **모든** 이름에 대해 키를 만들기 때문에, 힙에 이름이 있다는 것은 이 클라이언트가 그 FVariable 을 가졌다는 증거가 못 된다. 실제로 다음과 같은 이름이 있었다.

```
Pandora.exe --fflag-locate DFIntPhysicsSteppingMethod
  "PhysicsSteppingMethod" 문자열 자리 3개
    → 이 문자열을 이름으로 삼는 진짜 descriptor 노드가 없다
```

3.2 서술자 뜯어보기

후보 노드가 가리키는 서술자를 통째로 펼쳐 필드를 본다(`--fflag-desc`). 값이 `+0xC0` 에 있다는 것은 여기서 나온다. 같은 출력에서 `+0xD0` 에 (모듈 안 문자열 포인터, 길이) 짝이 보였고, 길이가 이름 길이와 정확히 같았다.

```
FIntDebugTextureManagerSkipMips    +0xD0 → 0x7FF684D8C048 ("Debu…")  +0xD8 = 27
DFIntTaskSchedulerTargetFps        +0xD0 → 0x7FF684CCC860 ("Task…")  +0xD8 = 22
DFFlagEnableZstdForClientSettings  +0xD0 → 0x7FF684E7EC80 ("Enab…")  +0xD8 = 27
```

서술자가 자기 이름을 들고 다닌다는 뜻이다. 이 한 줄이 신원 확인의 근거를 바꿨다(7.3절).

3.3 오프셋은 채점으로 확정한다

배치는 빌드마다 달라질 수 있으므로 상수로 박지 않는다. 이름을 이미 아는 인덱스 항목을 표본으로 삼아, 각 오프셋에서 읽은 이름표가 실제 이름과 맞는 비율을 센다(`--fflag-ident`).

```
Pandora.exe --fflag-ident
  표본 3000개에서 이름표가 발견된 오프셋:
    +0x00D0    1041개   ← 발견된 이름표의 96%
    (그 밖의 자리)  40개
  이름 일치 34.9% · 불일치 2.2% · 이름표 없음 62.9%
```

이름표가 3분의 1 에서만 나온다는 사실도 여기서 얻었다. 그래서 이 증거는 순위를 올리는 데만 쓴다.

3.4 실패를 vtable 로 묶는다

"몇 %를 넣을 수 있나"(`--fflag-coverage`)가 97.0% 로 나왔을 때, 남은 3%를 하나씩 보는 대신 vtable 로 묶었다. 같은 클래스의 서술자는 vtable 을 공유하므로, 묶으면 몇 가지 물건이 섞여 있는지가 한 번에 드러난다. 세 부류가 나왔고 셋 다 플래그가 아니었다(6.5절).

가짜 리스트는 값 충돌로 먼저 드러났다. 서로 무관한 플래그가 한 주소를 공유하고 있었다.

```
★ 값 충돌 — [desc 0x7FF6D54184C0]
   FFlagDebugSelfViewPerfBenchmark=false  vs  FFlagLuaMenuPerfImprovements=true
   vs FFlagUseNotificationServiceIsConnected=false  vs  FStringInExperienceNotificationsLayer=""  …
```

텔레메트리 서술자는 스트라이드로 드러났다. 이름을 자기 안에 `std::string` 으로 품고 `0xA0` 간격으로 놓여 있어서, `+0xC0` 을 값으로 읽으면 다음 원소의 한복판을 가리킨다.

```
TotalSwap     +0x18 "TotalSwap"  +0x38 "MemoryStats"  … 스트라이드 0xA0
AppVersion    +0x18 "AppVersion" +0x38 "Session"      … 스트라이드 0xA0
```

3.5 바깥 증거로 가른다

"없다"는 우리 인덱스의 주장이므로 인덱스 밖에서 확인한다. 플래그 이름은 모듈의 읽기 전용 데이터에 문자열 리터럴로 박혀 있으므로, 모듈 문자열표를 통째로 긁어 대조하면 우리가 틀렸는지 목록이 틀렸는지가 갈린다(`--fflag-why`).

```
━━ 바깥 증거 — 모듈 문자열표 122,920개 ━━
  이 빌드에 이름 자체가 없다    225개
  이름이 바뀐 것으로 보인다       2개
      DFFlagSendRenderFidelityTelemetry  →  SendRenderFidelityTelemetry2
      DFFlagClientRolloutPhaseTelemetry  →  ClientRolloutPhaseTelemetryFakeThrottle
  ★ 있는데 못 넣었다 (우리 몫)    0개
```

3.6 최적화는 차분으로 지킨다

스윕을 빠르게 만들 때마다 무엇을 잃었는지는 개수로 알 수 없다. 하나가 빠지고 하나가 더해져도 개수는 같다. 그래서 이름 집합을 통째로 견준다(`--fflag-scan-diff`, 10.6절). 없는 플래그 보고서는 이전 판의 출력과 바이트 단위로 비교한다.

3.7 크래시는 이분한다

원인을 모르는 종료가 하나 남아 있다. 종료 코드는 `0xE06D7363`(MSVC C++ 예외)이고, 로그의 마지막 줄은 언제나 120초 주기의 설정 폴링이라 원인을 말해 주지 않는다. 추측 대신 한 세션에 가설을 하나씩 지운다. 각 단계는 바로 앞 단계에 하나만 더한다.

단계	하는 일	지우는 가설
none	붙고 인덱스만 만든다(쓰기 0건)	읽기·스캔이 원인인가

표 1 — 크래시 이분 단계. 각 단계는 바로 앞 단계에 하나만 더한다.

resolve	이름까지 전부 푼다(쓰기 0건)	해석이 원인인가
notext	값이 std::string 으로 사는 슬롯을 뺀다(실측 약 80개)	텍스트 슬롯 쓰기가 원인인가
textonly	그 슬롯만 쓴다	위의 반대쪽
full	평소대로 전부 쓴다	기준선

한쪽만 돌리면 "빼도 죽는다"와 "빼면 산다"를 가를 수 없으므로 notext 와 textonly 를 모두 돌린다. 목록을 덩어리로 나눠 쓰고 덩어리마다 관찰하는 방식도 같은 원리이며, 죽으면 그 덩어리를 crash-subset.txt 로 남겨 다음 세션에서 다시 쪼갠다. 이 종료는 아직 원인을 확정하지 못했다.

4. 계층

계층	구성 요소	정하는 것
L0	MemorySource	프로세스 메모리 추상화: 읽기, 쓰기, 데이터로서 쓰기 가능 여부, 모듈 범위. 시험에서는 모의 구현으로 대체
L1	Knowledge / Operational	구조를 아는가와 써도 되는가를 분리하는 3상태 모델
L2	judge_registry	독립 증거 누적으로 레지스트리 배치를 판정
L3	TypeContract	bool·int·float·string 별 메모리 계약. 슬롯이 가져야 할 것과 가지면 안 되는 것
L4	apply_typed	트랜잭션 적용: 값이 바뀔 때만 쓰고, 쓴 뒤 다시 읽어 확인
L5	AuditTrail	단계별 거부 사유 기록

표 2 — 주입 경로의 계층과 각 계층이 정하는 것.

표 2 의 각 계층은 바로 아래 계층에만 의존한다. L0 를 모의 구현으로 바꾸면 프로세스 없이 L1–L5 전체를 시험할 수 있다. 실행 경로(프로세스 붙기, 인덱스 구축, 재적용 루프, 진단)는 이 위에 있다.

5. 메모리 접근

5.1 핸들과 권한

프로세스는 접근 마스크 0x0438(PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_QUERY_INFORMATION)로 연다[1]. 원격 스레드를 만들지 않고, 대상 프로세스에 메모리를 할당하지 않으며, 페이지 보호를 바꾸지 않는다. 쓰기는 이미 존재하는 값 위의 1~8바이트다. 이전 판에는 PROCESS_ALL_ACCESS 로 핸들을 얻고 대상 페이지를 PAGE_EXECUTE_READWRITE 로 바꾸는 코드가 있었으나 호출되지 않는 경로였고, 제거했다. 값을 쓰는 데 실행 권한은 필요 없다.

5.2 페이지 캐시와 실패 기억

읽기는 4KB 페이지 단위로 캐시한다. 캐시는 TTL 을 갖고, 한 번의 잠금으로 **필요한 바이트만** 복사한다. 이전 구현은 적중할 때마다 4KB 벡터를 통째로 복제했고, 인덱스를 한 번 짓는 동안 memcpy 총량이 4.26GB 였다.

읽기 실패도 같은 TTL 로 기억한다. 죽은 페이지를 반복해서 커널에 다시 묻던 구간에서 실패한 읽기가 30,970회로 전체 호출의 30% 였고, 실패한 왕복은 성공한 왕복과 같은 비용을 치른다. 실패 기억을 넣은 뒤 1,597회가 됐다. 페이지보다 큰 읽기(4페이지 초과)는 캐시를 거치지 않고 직접 읽는다.

스윕처럼 연속된 대량 읽기는 64KB 블록으로 호출한다. 같은 바이트를 4KB 단위로 읽는 것보다 ReadProcessMemory 호출 수가 16분의 1이다.

5.3 병렬 스윕과 순서 보존

레지스트리를 찾으려면 힙을 훑어야 한다. 훑기는 1MB 조각으로 나눠 여러 스레드에 분배하고, 조각 128개를 한 다발로 처리한다(실측: 다발 크기 32·64·128 이 모두 420ms 근처, 256 은 585ms). 조각마다 번호를 달아 결과를 모으므로 후보의 순서는 단일 스레드 구현과 같다. 순서는 성능이 아니라 정확성 문제다. 어느 리스트를 먼저 만나느냐가 어느 씨앗을 채택하느냐를 바꾸고, 과거에 이 순서가 흔들려 적용률이 80%에서 8%로 떨어진 적이 있다.

전역 페이지 캐시는 잠금이 하나라 여러 스레드가 함께 쓰면 그 잠금이 병목이 된다. 판별 단계의 스레드에는 잠금 없는 직접 사상 캐시를 따로 준다.

스레드 수의 실측 곡선은 1개 1,379ms, 2개 895ms, 4개 551ms, 6개 448ms, 8개 429ms 였다. 6에서 멈춘 것은 그 위로 얻는 것이 작고, 같은 기기에서 도는 게임 프로세스에서 가져가는 것만 늘기 때문이다.

5.4 영역 선택

--fflag-regions 로 영역 종류별 기여를 쟀다. 파일이 받쳐 주는 영역(MEM_MAPPED)은 바이트의 25%(503MB)를 차지하면서 후보의 1.4%만 주었고, 그 후보 중 리스트가 된 것은 하나도 없었다. 엔트리 노드는 이름 문자열을 힙에 들고 있는 C++ 객체이므로 CRT 힙(MEM_PRIVATE)에서 나온다.

이 관측을 최적화로 쓰되 믿고 넘어가지는 않는다. 건너뛴 실행의 결과 개수가 직전 실행보다 0.5% 넘게 모자라면 전부 읽는 방식으로 한 번 더 짓는다.

6. 레지스트리 발견

6.1 배치 모형

배치는 고정 상수가 아니라 추론 결과다. `MapLayout` 은 싱글턴에서 sentinel·버킷 배열·마스크까지의 오프셋, 버킷의 스트라이드와 체인 머리 위치, 엔트리의 next/name/descriptor 오프셋, 서술자의 값 오프셋을 담는다. 기본값은 지금까지 관측된 값이고(부록 B), 탐색이 갱신한다.

마스크 판정은 두 가지를 쓴다. 탐색용은 엄격하다. 값이 `2^n - 1` 꼴이고 `0x3F` 이상 `0x0100_0000` 미만이어야 한다. 플래그가 수천 개인 맵에서 버킷이 64개 미만일 수 없기 때문이다. 읽기용은 느슨하다. 이미 확정된 맵을 새 기준으로 다시 걷어차면 되던 것이 죽는다.

6.2 증거와 판정

증거는 표 3 의 다섯 가지이고 비트 집합으로 누적한다.

증거	내용	종류
`PointerGraph`	버킷 배열 포인터가 유효한 사용자 주소	구조적
`HashMask`	마스크가 `2^n - 1` 꼴	구조적
`ChainWalk`	체인을 걸어 둘 이상의 노드를 읽음	구조적
`NameString`	노드에서 읽은 문자열이 플래그 이름으로 타당	의미론적
`TypedValueSlot`	서술자가 가리키는 값 슬롯이 이름이 말하는 타입과 일치	의미론적

표 3 — 레지스트리 판정에 쓰는 독립 증거.

판정 규칙은 하나다. **독립 증거 두 개 이상이고, 그중 하나는 반드시 의미론적 증거여야 한다.** 구조적 증거만 모으면 그것은 "포인터와 작은 수를 담은 아무 객체"다. 규칙을 통과하지 못하면 `Knowledge::Unknown` 으로 남고, 기본 상태가 쓰기 금지이므로 아무 일도 일어나지 않는다.

```
pub fn verdict(&self) -> Knowledge {
    let semantic = self.has(Evidence::NameString) || self.has(Evidence::TypedValueSlot);
    if self.count() >= 2 && semantic {
        Knowledge::Valid
    } else {
        Knowledge::Unknown
    }
}
```

목록 1 — 판정 규칙(`fflag_engine.rs`). `count()` 는 누적된 증거 비트 수, `semantic` 은 `NameString` 또는 `TypedValueSlot` 의 존재다. 둘을 함께 요구하는 것이 이 규칙의 전부다.

6.3 후보 거르개와 확인

훑기 단계는 노드 후보를 모으고, 확인 단계는 후보에서 리스트를 세운다. 확인이 반드시 요구하는 조건(링크가 `+0x00`/`+0x08`, 서술자 포인터가 `+0x30`~`+0x48`)을 스캔 버퍼 안에서 미리 보게 하면 후보가 155,218개에서 107,501개로 줄고 확인이 391ms 에서 228ms 가 된다. 이미 손에 든 바이트를 보는 것이라 읽기는 늘지 않는다.

확인은 16,384개씩 나눠 돌린다. 판별은 읽기만 하고 서로 독립이라 스레드로 나누고, 리스트 걷기는 이미 채택된 노드 집합을 공유하므로 순차로 둔다. 나눠 도는 이유는 큰 리스트가 일찍 잡히면 그 리스트에 속한 노드가 뒤 덩어리의 판별에서 통째로 빠지기 때문이다(실측: 후보의 4분의 1).

조기 종료는 한 번 넣었다가 되돌렸다. 주 레지스트리로 볼 만큼 큰 리스트(16,000개 이상)가 나오면 거기서 멈추는 최적화였다. 이름의 71%가 한 레지스트리에 있어 합리적으로 보였지만, 일찍 멈춘 실행은 레지스트리를 11~15개만 찾았고 전부 읽으면 100개가 넘었다. 결과가 실행마다 달라졌다. 적용 수가 흔들리는 것은 느린 것보다 나쁘다. 어느 쪽이 맞는 값인지 판단할 근거가 사라지기 때문이다.

6.4 리스트가 레지스트리인지 확인한 뒤에 합친다

각 리스트는 따로 걸어 이름→서술자 사상을 만든 뒤 합친다. 이름이 8개 이상인데 서로 다른 서술자 수가 이름 수의 4분의 1 미만이면 그 리스트는 버린다.

이 규칙은 관측에서 나왔다. 노드 배치를 잘못 짚어 모든 노드가 같은 자리를 가리키는 리스트가 있었고(이름 4,339개, 서술자 1개), 합쳤을 때 서로 무관한 플래그 열아홉 개가 같은 주소로 해석됐다. 그 열아홉 개는 한 자리에 서로 다른 값을 차례로 덮어쓰면서 각자 "적용됨"으로 세어졌다. 더 나쁜 것은 접두사 붙은 변형이 먼저 걸리는 바람에 진짜 서술자를 가진 기본 이름은 시도조차 되지 않은 점이다. 진짜 레지스트리는 이름과 서술자가 1:1 이다.

```
let mut one: HashMap<String, Vec<u64>> = HashMap::new();
merge_entry_list_multi_ex(&mem, *seed, *el, &mut one, 400_000, &mut rejected);

let distinct: HashSet<u64> = one.values().flatten().copied().collect();
if one.len() >= 8 && distinct.len() * 4 < one.len() {
    continue;
}
```

목록 2 — 리스트를 합치기 전의 검사(`fflag_injector.rs`). 리스트를 따로 걸어 이름→서술자 사상을 만든 뒤, 서술자 기수가 이름 기수의 4분의 1 미만이면 전역 인덱스에 합치지 않고 버린다.

6.5 인덱스 오염

`--fflag-coverage` 로 잰 주입 가능 비율이 97.0% 였다. 남은 3%를 vtable 로 묶으니 세 부류였고 셋 다 플래그가 아니었다.

1. **Roblox 인스턴스**(204개 중 189개). vtable 이 힙 주소였다. 실제 서술자는 정적으로 놓인 클래스라 vtable 이 모듈 이미지 안에 있다. 사용자 플래그 이름이 우연히 겹치면 인스턴스에 쓰게 되며, 이는 2.2절의 크래시 경로다.
2. **텔레메트리 통계 서술자**. 이름을 자기 안에 `std::string` 으로 품는다. 실측한 64개 전부가 그랬고 인덱스의 0.97% 였다. 스트라이드가 `0xA0` 인데 값은 `+0xC0` 에서 찾으므로 그 자리는 다음 원소의 한복판이고, 64개 중 51개가 "값 슬롯이 있다"로 보였다. 값이 있어 보이는 쪽이 더 위험하다.
3. **가짜 리스트**(6.4절).

세 부류를 제거한 결과는 10.4절에 있다.

7. 이름에서 슬롯까지

7.1 이름 변형

사용자 목록의 이름이 클라이언트 내부 이름과 다를 수 있다. 접두사를 잘못 알고 있거나, 이름이 바뀐 경우다. 그래서 한 이름에 대해 변형을 만든다. 원본, 접두사를 뗀 기본 이름, 같은 타입의 다른 접두사, 모든 접두사, 흔한 오타 교정 순이다. 한 플래그당 변형은 최대 28개이고, 변형 하나마다 주소 해석이 다시 14개까지 후보를 만든다.

이 탐색은 값이 싸지 않다. 없는 이름 하나를 확인하는 데 해시 조회와 소문자 변환이 수백 번 발생한다. 7.4절의 확정 기억이 이 비용을 한 번으로 만든다.

7.2 서술자 후보와 값 슬롯

한 이름이 여러 서술자를 가리킬 수 있다. 접두사를 뗀 키가 같은 쌍이 인덱스에 6,563개 있다. 후보가 여럿이면 값 슬롯이 이름이 말하는 타입과 맞는 쪽을 고르고, 그래도 갈리면 7.3절의 이름표를 가진 쪽을 앞세운다. 후보가 하나뿐이면 값 슬롯을 열어 보지 않는다. 하나면 답이 정해져 있기 때문이다. 인덱스 41,039개 중 36,821개가 이 경우이고, 이 생략으로 후보 고르기가 168ms 에서 31ms 가 됐다.

값 슬롯을 찾지 못하면 다른 오프셋을 훑는데, 그 목록에 `+0xD0` 이 들어 있었다. 이름표를 값 슬롯으로 집고 있었다는 뜻이다. 그 자리는 읽기 전용 구역이라 쓰기는 실패하지만, 해당 플래그는 버려지고 실패가 "레지스트리가 다시 지어졌다"로 오인돼 인덱스를 통째로 버리게 만든 사례가 있다 (실측: 2차 주입이 통째로 건너뛰어졌다). 지금은 그 자리가 이름표이면 값 슬롯 후보에서 뺀다.

7.3 서술자의 자기 이름

"이 주소가 아직 그 플래그인가"를 vtable 이 모듈을 가리키는지로 묻는 것은 약하다. 해제된 힙은 옛 vtable 을 그대로 담고 있고, 다른 플래그의 서술자도 같은 vtable 을 갖는다. 이름은 다르다.

서술자는 `+0xD0`에 (모듈 내 문자열 포인터, 길이) 짝을 들고 있다. 이름표 후보로 받아들이는 조건은 길이가 1~192 이고, 포인터가 주 모듈 범위 안이며, 그 자리를 읽었을 때 길이만큼의 문자열이 나오는 것이다. 이름 문자열은 읽기 전용 구역에 있어 프로세스가 사는 동안 이동하지 않으므로, 이 16바이트 짝을 한 번 기억하면 그 뒤로는 8바이트 비교 두 번이 곧 신원 확인이 된다.

오프셋은 하드코딩하지 않는다. 실측에서 얻은 후보(`+0xD0`)를 먼저 보고, 맞지 않으면 나머지를 훑되, 채택은 이름을 이미 아는 인덱스 항목으로 채점해서 한다. 표본 3,000개에서 이름표가 발견된 오프셋의 96%가 `+0x00D0` 였다(1,041개). 같은 표본에서 이름 일치는 34.9%, 불일치 2.2%, 이름표 없음 62.9% 였다.

```
fn name_pair_at(h: HANDLE, desc: u64, off: u64, mbase: u64, msize: u64) -> Option<String> {
    let p = ru64(h, desc + off);
    let n = ru64(h, desc + off + 8);
    if !(1..=192).contains(&n) { return None; }
    if mbase != 0 && (p < mbase || p >= mbase.saturating_add(msize)) { return None; }
    if !is_ptr(p) { return None; }
    let b = rmem(h, p, n as usize)?;
    if b.len() < n as usize { return None; }
    if !b.iter().all(|c| c.is_ascii_alphanumeric() || *c == b'_') { return None; }
    if !b[0].is_ascii_alphabetic() { return None; }
    Some(String::from_utf8_lossy(&b).into_owned())
}
```

목록 3 — 이름표 후보 판정(`fflag_injector.rs`). 길이 1–192, 포인터가 주 모듈 이미지 범위 안, 역참조 성공, ASCII 식별자 형태를 모두 만족해야 이름표로 받아들인다. 힙 포인터는 여기서 떨어진다 — 이름 리터럴은 `.rdata` 에 있다.

이름표가 있는 서술자는 3분의 1 뿐이므로 이 증거는 후보의 순위를 올리는 데만 쓰고, 없다고 해서 떨어뜨리지 않는다. 증거가 없는 것은 반증이 아니다. 이 구분을 놓쳐 생긴 오류가 실제로 있었다. 인덱스를 짓지 않은 프로세스에서 되돌리기를 하면 이름표 오프셋을 배우지 못하는데, 그 상태를 "이름표가 달랐다"로 읽어 멀쩡한 슬롯 두 개를 모두 "자리 옮김"으로 버렸다. 지금은 되돌리기 전에 인덱스를 지어 배우게 하고, 그래도 배우지 못하면 vtable 검사로 내려간다.

7.4 없음의 확정과 폐기

인덱스가 이 빌드의 전수 목록이면 "없다"는 조회 실패가 아니라 확정이다. 확정은 기억한다. 단, 세 가지 조건을 건다.

- 인덱스 항목이 512개 미만이면 기억하지 않는다. 그 인덱스는 전수 목록이라고 볼 수 없다.
- 인덱스가 바뀌면 세대 번호가 오르고 장부는 통째로 폐기된다.
- 인덱스가 오래 그대로여도 5분마다 한 번은 장부를 비운다. 프로세스가 바빠 읽기가 한 번 빗나간 것을 "없음"으로 박으면 그 플래그는 영영 들어가지 않는다. 비용은 5분에 한 번의 재탐색이다.

이 기억으로 이름 해석과 쓰기가 22.8ms 에서 0.6ms 가 됐다(없는 이름 227개 기준).

8. 쓰기 경로

8.1 3상태 모델

`Knowledge` 는 구조에 대한 앎(`Valid`/`Invalid`/`Unknown`)이고, `Operational` 은 행위에 대한 허가(`ReadOnly`/`WriteCandidate`/`WriteBlocked`)다. 둘을 분리한 것이 이 엔진의 뼈대다. 구조를 확정했으면서 쓰지 않는 상태(예: 정책상 읽기 전용인 문자열)가 정상적으로 존재한다.

기본값은 `Unknown` + `WriteBlocked(UnknownStructure)` 다. 증거가 이 상태를 끌어올리지 못하면 그 슬롯은 건드리지 않는다.

거부 사유(`BlockReason`)는 열거형이고 감사의 1차 자료다. 구조 미상, 타입 계약 위반(관측값 포함), 쓰기 불가 메모리, 문자열 capacity 초과, 정책상 읽기 전용, 표현 불가능한 값, 읽기 실패, 서술자 아님, 자해, 정렬 어긋남이 있다. 타입 계약 위반은 반증의 근거가 된 실제 관측값(`Observed`)을 함께 싣는다. bool 자리가 0/1 이 아니었다, int 자리가 포인터로 보였다, float 자리가 유한수가 아니었다, `std::string` 으로 파싱되지 않았다 중 하나다.

8.2 타입 계약

계약은 슬롯이 가져야 할 조건과 **가지면 안 되는** 조건을 함께 명시하고, 같은 파일 안에서 검증· 인코딩·비교·읽기를 한 벌로 묶는다. 되돌리기가 읽는 규칙과 적용이 쓰는 규칙이 조금이라도 어긋나면 되돌린 값이 원래 값이 아니게 되기 때문이다.

bool. 1바이트를 읽어 0 또는 1 이 아니면 반증한다. 잘못된 주소이거나 함정이다.

int. int 는 어떤 4바이트든 그럴듯하므로 자기 검증이 약하다. 그래서 명백히 int 가 아닌 경우만 반증한다. 세 조건이 모두 맞아야 반증이다. (a) 8바이트가 포인터 모양(사용자 주소 범위, 상위 32비트 살아 있음, 8바이트 정렬)이고, (b) 덮어쓸 하위 4바이트 자체가 FInt 값으로 보기에 지나치게 크며 (절댓값 `0x1000_0000` 초과), (c) 그 포인터를 실제로 따라갈 수 있다.

세 조건이 모두 필요한 이유는 배치에 있다. `FInt` 값들은 `.data` 에 4바이트씩 인접해 놓이므로 값 주소에서 8바이트를 읽으면 상위 4바이트는 이웃 플래그의 값이다. 이웃 값이 1~32767 이고 우리 값이 8의 배수이면 그 8바이트는 포인터의 모양이 된다. (a) 하나로 거부하면 멀쩡한 int 슬롯이 그 우연으로 버려지고, 다음 후보 오프셋으로 밀려가 이름표를 값 슬롯으로 집는다. 실제로 그랬다. 이 판정은 한 함수에만 둔다. 같은 판정이 두 군데 있으면 반드시 어긋나고, 어긋난 쪽이 플래그를 잃거나 크래시를 낸다.

```
pub fn looks_like_pointer(q: u64) -> bool {
    is_user_ptr(q) && (q >> 32) != 0 && (q & 0x7) == 0
}

pub fn int_slot_is_plausible(low: i32, q: u64, deref_ok: bool) -> bool {
    !(looks_like_pointer(q) && (low as i64).abs() > 0x1000_0000 && deref_ok)
}
```

목록 4 — int 슬롯 반증 규칙(`fflag_engine.rs`). `low` 는 덮어쓸 하위 4바이트, `q` 는 같은 주소에서 읽은 8바이트, `deref_ok` 는 그 8바이트를 주소로 보고 역참조에 성공했는지다. 반증은 셋이 모두 맞을 때만 성립하며, 이 규칙은 코드 전체에서 이 함수 하나뿐이다.

인코딩에서 범위를 넘는 값은 자르지 않고 조인다. `as i32` 는 하위 32비트만 남기는 순환이라, 크게 넣으려고 적은 수가 임의의 값이 된다. 실측에서 `9999999999999999` 가 `0x6FC0FFFF`(1874919423)로 들어가 있었다. 비교(`equals_current`)도 같은 규칙으로 조여야 한다. 아니면 값이 맞는데도 매 주기 다시 쓰고, 쓴 뒤 비교가 영원히 실패해 "게임이 되돌렸다"로 오진된다.

float. 4바이트를 `f32` 로 읽어 유한수가 아니면 반증한다.

```
pub fn read_std_string(m: &dyn MemorySource, sp: u64, max: usize) -> Option<String> {
    let d = m.read(sp, 0x20)?;
    let size = u64::from_le_bytes(d[0x10..0x18].try_into().ok()?) as usize;
    let cap  = u64::from_le_bytes(d[0x18..0x20].try_into().ok()?) as usize;
    if cap < 15 || cap > 0x10_0000 || size > cap || size > max {
        return None;
    }
    if size == 0 {
        return Some(String::new());
    }
    let buf = if cap > 15 { u64::from_le_bytes(d[0x00..0x08].try_into().ok()?) } else { sp };
    if !is_user_ptr(buf) { return None; }
    let raw = m.read(buf, size)?;
    if raw.len() < size { return None; }
    String::from_utf8(raw).ok()
}
```

목록 5 — MSVC `std::string` 읽기(`fflag_engine.rs`). `+0x00` 은 16바이트 union(capacity 가 15 를 넘으면 힙 포인터, 아니면 SSO 버퍼), `+0x10` 은 size, `+0x18` 은 capacity 다. `size == 0` 은 실패가 아니라 빈 문자열이다. 이 배치는 코드 전체에서 이 함수 하나에만 적혀 있다. 예전에 `+0x08`=len, `+0x10`=cap 으로 알고 있던 코드가 문자열 플래그를 파괴했다.

string. `std::string` 으로 파싱되는지가 곧 검증이다(capacity 15 이상 `0x10_0000` 이하, size ≤ capacity). 이 검증은 강한 신호라 정책상 읽기 전용이어도 레지스트리 판정의 증거로 쓸 값어치가 있다. 쓰기는 capacity 를 넘지 않을 때만 한다. 재할당은 흉내 낼 수 없다. 쓰는 대상은 슬롯이 아니라 버퍼이고(capacity 15 초과면 힙 포인터, 이하면 객체 내부), size 필드만 함께 갱신하며 capacity 는 건드리지 않는다.

여기에 한 칸짜리 오류가 있었다. 조건이 `길이 + 1 > capacity` 였는데, MSVC 의 `_Myres` 는 쓸 수 있는 글자 수이고 버퍼는 `_Myres + 1` 바이트다. 길이가 capacity 와 같은 값은 널 종단까지 정확히 들어간다. 그 한 칸 때문에 15글자짜리 값이 버려지고 있었다.

빈 문자열도 정상 값이다. 이것을 읽기 실패로 처리하던 시절에는 문자열 플래그를 `""` 로 두는 것이 원리적으로 불가능했다. 쓰기는 성공하는데 읽기 후 비교가 `None != Some("")` 이 되어 매번 "게임이 되돌렸다"로 오진됐다. 프로세스를 정지시킨 채 써도 같은 결과가 나온 것이 증거였다. 멈춘 프로세스는 되돌릴 코드를 실행할 수 없으므로 되돌린 쪽은 게임이 아니었다.

8.3 원자성

스칼라 쓰기는 정렬됐을 때만 발행한다(2·4·8바이트는 크기로 나누어떨어지는 주소, 1바이트는 항상 허용). 정렬이 어긋난 쓰기는 쪼개질 수 있고, 그러면 게임이 읽는 값은 옛 값도 새 값도 아닌 섞인 비트가 된다. 우리가 쓴 적 없는 수이므로 어떤 검증으로도 예측할 수 없고 재현도 타이밍에만 걸린다. 검사는 산술 한 줄이고, 통과하지 못하는 자리는 애초에 그 타입의 슬롯이 아니다.

8.4 트랜잭션

`apply_typed` 의 순서는 다음과 같다.

1. **자해 금지.** 클라이언트가 자기 설정을 읽지 못하게 만드는 값은 먼저 막는다. 타입도 주소도 정상이라 아래 검사는 모두 통과하며, 이 쓰기는 성공했을 때 게임을 죽인다.
2. **읽기 우선 검증.** 계약의 `verify`. 반증이면 관측값과 함께 중단한다.
3. **정책.** 타입 자체가 쓰기 대상이 아니면 사유와 함께 중단한다.
4. **변화 없으면 쓰지 않는다.** 현재 값이 목표와 같으면 `WriteProcessMemory` 자체를 발행하지 않는다.
5. **인코딩.** 표현할 수 없는 값이면 중단한다.
6. **원자성**(8.3절)과 **쓰기 가능성**. 실행 페이지나 미커밋 페이지면 중단한다.
7. **쓰기.** 문자열은 버퍼에 쓰고 size 필드를 함께 갱신한다.

8. **쓴 뒤 읽기.** 같은 계약으로 다시 비교한다. 값이 되돌아가 있으면 실패가 아니라 경쟁 상태로 보고 다음 주기에 다시 시도한다.

```
pub fn apply_typed(m: &dyn MemorySource, slot: u64, ty: FlagType,
                   value: &FlagValue, flag_name: &str) -> TxOutcome {
    if let Some(why) = self_harm_reason(flag_name, value) {
        return TxOutcome::Blocked(BlockReason::SelfHarm(why));
    }
    let c = ty.contract();

    match c.verify(m, slot) {
        SlotVerdict::Consistent => {}
        SlotVerdict::Refuted(o) => return TxOutcome::Refuted(o),
        SlotVerdict::Unreadable => return TxOutcome::Blocked(BlockReason::Unreadable),
    }
    if !c.writable() { return TxOutcome::Blocked(BlockReason::PolicyReadOnly(ty)); }
    if c.equals_current(m, slot, value) { return TxOutcome::Unchanged; }

    let enc = match c.encode(m, slot, value) {
        Ok(e) => e,
        Err(r) => return TxOutcome::Blocked(r),
    };
    if !scalar_write_is_atomic(enc.addr, enc.bytes.len()) {
        return TxOutcome::Blocked(BlockReason::Misaligned {
            addr: enc.addr, len: enc.bytes.len(),
        });
    }
    if !m.is_writable_data(enc.addr, enc.bytes.len()) || !m.write(enc.addr, &enc.bytes) {
        return TxOutcome::Blocked(BlockReason::NotWritableMemory);
    }

    if c.equals_current(m, slot, value) { TxOutcome::Committed } else { TxOutcome::Recomputed }
}
```

목록 6 — 트랜잭션의 뼈대(`fflag_engine.rs`). 순서가 곧 계약이다. 자해 차단 → 읽기 우선 검증 → 정책 → write-on-change → 인코딩 → 원자성 → 쓰기 가능성 → 쓰기 → read-after-write. 감사 기록(`audit_push`)과 문자열 `size` 필드 갱신은 지면상 줄였다.

결과는 `Committed`/`Unchanged`/`Recomputed`/`Refuted`/`Blocked` 다. 화면의 "몇 개 적용"은 이 중 어디에 속하는지로 정의된다.

8.5 감사

거부는 단계(계약/트랜잭션)와 사유, 주소, 플래그 이름을 함께 기록한다. 실패 사유 수집은 평소에는 꺼 둔다(문자열을 만드는 비용이 있다). `--fflag-why` 가 한 번의 실행에서만 켠다.

한 플래그는 변형을 여럿 시도하고 대부분은 "그런 이름 없음"으로 끝나므로, 흔한 사유가 진짜 사유를 덮으면 표가 쓸모없어진다. 그래서 사유에 "얼마나 멀리 갔는가"의 등급을 매기고 더 멀리 간 쪽만 덮어쓴다.

9. 상태 유지

9.1 되돌리기

되돌리기의 정의는 하나다. 우리가 쓰기 직전에 그 자리에 있던 값을 다시 쓴다. 빌드 기본값을 추측하지 않는다. 그 값은 서버가 내려준 것일 수도 있고, 구현이 아는 값이 아니다.

모든 쓰기가 지나는 한 지점에서 그 슬롯에 **처음** 쓰기 직전의 값을 장부에 남긴다. 두 번째부터는 우리가 쓴 값이라 원본이 아니다. 장부의 머리에는 PID, 프로세스 생성 시각, 모듈 베이스, 빌드 버전을 적는다. Windows 는 PID 를 곧바로 재사용하므로 PID 만으로는 같은 프로세스임을 보장하지 못한다. 넷 중 하나라도 다르면 장부는 재생되지 않고 폐기된다. 항목에는 값 주소, 서술자 주소, 이름표 (포인터, 길이), 타입, 이전 값을 함께 적는다.

되돌리기 전에 세 가지를 확인하고, 하나라도 어긋나면 그 슬롯은 건드리지 않는다.

물음	확인 방법	어긋났을 때
그 자리가 아직 그 플래그인가	이름표 16바이트 비교	자리 옮김 — 힙은 재사용된다
값이 아직 우리가 쓴 값인가	쓰기와 같은 계약의 비교	게임이 바꿈 — 지금 값이 깨끗한 값이다
써도 되는 자리인가	`apply_typed` 전체	실패 — 장부에 남겨 다시 시도

표 4 — 되돌리기 전에 확인하는 세 가지.

두 번째가 중요하다. 우리가 쓴 뒤 클라이언트가 스스로 값을 바꿨다면 그 자리의 깨끗한 상태는 장부의 옛 값이 아니라 현재 값이다. 옛 값을 넣는 것은 되돌리기가 아니라 새로운 주입이다.

되돌리는 동안 자동 적용과 재적용은 한 주기를 통째로 쉰다. 그러지 않으면 되돌린 값이 2초 뒤 다시 덮여, 화면은 되돌렸다고 말하는데 게임은 그대로인 상태가 된다.

되돌리지 못하는 것도 그대로 센다. 시작 시점에만 읽히는 플래그는 값을 제자리에 돌려놔도 화면이 즉시 돌아오지 않는다. 프로그램을 켜기 전부터 들어가 있던 플래그는 원래 값을 아무도 모르므로 장부가 비어 있다고 보고하고 추측해서 쓰지 않는다. 시작 시점 파일은 백업이 있으면 되살리되, 우리가 쓴 뒤 누군가 그 파일을 바꿨으면 손대지 않는다.

한 바퀴 실측은 다음과 같다. 값 두 개를 바꿔 적용하고, `--fflag-revert` 가 2개를 되돌렸으며(이미 원래 값 0, 게임이 바꿈 0, 자리 옮김 0, 실패 0), 다시 적용했을 때 같은 두 자리가 같은 이전 값으로 장부에 잡혔다. 마지막 단계가 증거다. 아무것도 쓰지 않았다면 그 적용은 `Unchanged` 로 끝나 장부에 아무것도 남지 않는다.

9.2 무효화

클라이언트 갱신은 레지스트리를 옮긴다. 상태 표시줄이 그 조건을 보고하고, 오프셋은 30초 → 1분 → 2분 → 5분 의 백오프로 다시 유도한다. 하나도 들어가지 않은 적용은 오프셋이 낡았다는 증거로 보고 갱신을 예약한 뒤, 갱신이 끝나면 한 번 다시 시도한다.

서버 이동은 더 급하다. 메모리 표본을 다시 확인해 절반이 한꺼번에 어긋나는지로 추측할 수 있지만, 추측이 맞는 시점은 이미 한 주기를 쓴 뒤이고, 그 주기가 해제된 힙에 쓰는 주기일 수 있다. 클라이언트는 그 사실을 로그에 스스로 적는다. 로그의 새로 붙은 꼬리만 읽고(처음 볼 때는 기준선만 잡는다), `submitStartGameTask` 표식이 보이면 그 주기는 쓰지 않고 인덱스·싱글턴·페이지 캐시를 버린 뒤 백오프를 건너뛰고 그 자리에서 다시 짓는다. 같은 감시기가 `RBXCRASH:` 와 순간이동 시작 표식도 읽는다.

합성 로그로 확인한 동작은 한 주기(2초)를 쉬고 다음 주기에 이전 수치로 복귀하는 것이었다. 인덱스 구축이 359ms이므로 복귀가 싸다. 실제 서버 이동 중의 동작은 명령줄에서 일으킬 수 없어 재현하지 못했다. 메모리 기반 추측은 로그를 읽지 못하는 경우(경로가 다르거나 로그가 갈린 경우)를 위해 함께 남겨 두었다.

9.3 자가 격리 장부

크래시를 본 뒤 원인을 좁히는 장부가 있다. 세션마다 이름별로 두 숫자(있었던 세션 수, 죽은 세션 수)만 세며, 죽음을 두 번 이상 보고 용의자가 3개 이하로 좁혀졌을 때 격리한다. 죽음·생존의 판정은 시계가 아니라 클라이언트 로그가 한다.

여기에 무작위 델타 디버깅("흔들기")이 붙어 있었다. 매 회차 용의자의 절반을 빼서 다음 세션의 결과에 정보를 싣는 기법이고, 좁은 집합에서는 유효하다. 문제는 넓은 집합이었다. 죽음 1회·용의자 191개 상태에서 이 장부는 사용자 플래그 76개 중 27개(35%)를 매 주기 제외하고 있었고, 상태 화면은 격리 목록만 보여 주어 그 비용이 드러나지 않았다. 191개를 3개까지 좁히려면 죽음을 여섯 번 더 봐야 하는데 실측 크래시율은 39세션에 1회다.

넓을 때의 올바른 수는 기다리는 것이다. 산 세션에 한 번이라도 있었던 이름은 용의자에서 자동으로 빠지므로 아무것도 빼지 않아도 집합은 줄어든다. 흔들기에 조건을 걸었다(죽음 2회 이상, 용의자 64개 이하). 그리고 상태 화면이 격리가 아니라 실제로 제외되는 이름을 보여 주게 했다. 302개 표의 적용은 48개에서 75개로 늘었고, 늘어난 27개가 흔들기가 빼고 있던 그 27개였다.

10. 평가

10.1 환경과 방법

모든 수치는 단일 기기에서 벽시계로 측정했고, 각 수치는 부록 A 의 명령으로 재현된다. 실험에 쓴 목록은 302개짜리 표와 715개짜리 별도 목록 두 가지다. 실행 시점과 구현 판이 다르므로, 서로 다른 실험의 개수를 더하거나 빼서 비교하지 않는다.

10.2 적용 결과

`--fflag-audit` 는 아무것도 쓰지 않고 메모리를 다시 읽어 요청한 값과 대조한다. 302개 표에 대한 결과는 다음과 같다(2026-09-05 기록).

항목	개수
일치	75
값이 다름	0
값 슬롯 없음	0
이 빌드에 이름이 없음	227

표 5 — 302개 표에 대한 읽기 전용 감사(`--fflag-audit`, 2026-09-05).

표 5 에서 이 빌드가 가진 이름 75개 중 75개가 요청한 값을 들고 있다.

별도의 715개 목록에서는 `--fflag-applied` 가 메모리를 다시 읽어 만든 목록이 423개였고, 이는 구현의 내부 카운터와 정확히 일치했다. 값이 다른 4개는 모두 사용자 목록 안의 중복 키였다(예: `FIntRenderShadowmapBias=0` 과 `RenderShadowmapBias=75`). 접두사를 뗀 키가 같으므로 둘은 같은 슬롯이고, 둘 다 넣을 수는 없다.

10.3 넣지 못한 이름의 외부 검증

"이 빌드에 없다"는 인덱스의 주장이므로 인덱스 밖에서 확인해야 한다. 증거를 둘 썼다.

첫째, 모듈 문자열표다. 플래그 이름은 모듈의 읽기 전용 데이터에 문자열 리터럴로 박혀 있다. 문자열표 122,920개와 대조한 결과 넣지 못한 227개 중 225개는 이 빌드에 이름 자체가 없었고, 2개는 뒤에 숫자가 붙은 후속 이름으로 바뀐 것이었다(`DFFlagSendRenderFidelityTelemetry` → `SendRenderFidelityTelemetry2` 등). 구현의 책임으로 남은 항목은 0개였다.

둘째, Roblox 의 공개 설정표(22,384개 이름)다. 별도의 715개 목록에서 없는 것으로 판정한 269개는 공개 설정표에서 하나도 발견되지 않았고, 같은 조회로 대조군 444개 중 245개가 발견됐다. 조회 자체가 고장이라면 양쪽을 모두 놓쳤을 것이다.

다만 설정 파일에 이름이 있다는 것이 이 빌드에 그 플래그가 등록돼 있다는 증거는 아니다. 서버 설정표는 서버가 보낸 모든 이름에 대해 키를 만들기 때문이다. 이 기기의 다른 출처도 같은 한계를 갖는다. `StudioAppSettings.json` 의 24,518개 중 이 빌드가 아는 것은 5,132개뿐이었다. 개별 이름은 `--fflag-locate` 로 직접 확인한다. 이름 문자열을 힙에서 찾고, 그것을 이름으로 삼는 `std::string` 헤더를 찾고, 그 헤더를 품은 노드가 실제 서술자를 들고 있는지 본다.

10.4 인덱스 오염 제거

	전	후
전체 플래그 대비 주입 가능	97.0%	표본 10,242개 중 10,240개
인덱스 크기	34,027	30,724

표 6 — 플래그가 아닌 항목 3,300개를 제거하기 전후.

제거한 3,300개는 6.5절의 세 부류다.

10.5 지연

같은 기기, 같은 Roblox 프로세스(상주 2.0GB)에서 실행 후 첫 플래그가 들어갈 때까지를 네 단계로 나눠 쟀다.

단계	전	후
데몬 첫 순찰 대기	2,000 ms	0 ms
인덱스 구축	3,013 ms	880 ms
첫 적용	2,981 ms	130 ms
전체	약 8 s	약 1.0 s

표 7 — 실행 후 첫 플래그까지의 단계별 시간. 같은 기기, 같은 프로세스(상주 2.0GB).

같은 구간에서 적용 결과는 715개 중 435개로 모든 단계에서 동일했고 인덱스 이름 집합도 같았다. 게임 안(상주 3.1GB)에서 잰 이전 판의 값은 인덱스 5,800ms, 첫 적용 6,119ms 였다.

원인별로는 5.2절(페이지 캐시 복제 4.26GB, 실패 읽기 30,970회), 5.3절(단일 스레드 스윕), 7.2절 (후보가 하나인데 메모리를 읽던 것, 168 → 31ms), 7.4절(없는 이름 재탐색, 22.8 → 0.6ms)이 각각 기여했다. 여기에 두 가지가 더 있다.

- 없는 플래그 보고서가 한 주기 적용의 80% 였다. 바뀌지 않은 목록에 대해 1,150만 번을 비교했고 195~299ms 가 걸렸다. 넣는 일 자체는 25~45ms 다. 이름을 정렬해 두면 접두 구간은 연속이므로 이분 탐색으로 닿고, 목록이 그대로면 계산하지 않는다.
- 없는 플래그를 만날 때마다 인덱스 재구축을 동기로 돌렸다. 1,390ms 를 쓰고 이름 3개를 얻은 경우도 있었다. 지금은 배경에서 짓고 완성된 뒤 한 번에 바꿔 끼운다. 처음에는 배경 스레드로 옮기기만 했는데 적용 시간이 그대로였다. 재구축이 살아 있는 인덱스를 먼저 비워 그동안의 조회가 모두 빗나갔기 때문이다. 짓는 일과 거는 일을 분리해야 했다.

디스크 쪽도 같은 원리로 줄였다. 학습 표·발견 목록·없는 목록은 내용 해시가 같으면 쓰지 않고, `flags.json` 은 (수정 시각, 크기)가 같으면 다시 파싱하지 않는다. 2초 주기에서 이 호출은 하루 약 4만 번이다.

10.6 검산과 안정성

더 빠른 스윕이 조용히 무언가를 잃었는지는 개수로 알 수 없다. 하나가 빠지고 하나가 더해져도 개수는 같다. 이름 집합을 통째로 비교했다.

```
Pandora.exe --fflag-scan-diff
  전부 읽기  41,042개
  빠른 판    41,042개
  전부에만 있는 이름 0개
```

느슨한 옛 거르개와도 같은 방식으로 비교했고 정렬한 이름 목록이 한 줄도 다르지 않았다. 없는 플래그 보고서도 이전 판의 출력과 바이트 단위로 같았다.

반복 실행의 안정성은 다음과 같다.

```
Pandora.exe --fflag-soak 90 2
  90판 · 매 판 75/302 · 적용 수 범위 75~75 · 프로세스 생존 · 표 재건 오탐 0회

Pandora.exe --fflag-soak 60 2
  60판 · 매 판 423/715 · 적용 수 범위 423~423 · 한 판 12.6~16.3 ms
```

적용 수가 주기마다 흔들리던 원인은 둘이었고 둘 다 제거했다. 하나는 6.3절의 조기 종료다. 다른 하나는 같은 슬롯을 가리키는 중복 키를 주소로 판정한 것이다. 주소는 캐시가 채워진 다음 주기에야 알 수 있으므로 첫 주기에는 둘 다 쓰고 그다음 주기부터 하나를 건너뛰었다(432 → 423). 접두사를 뗀 이름과 타입이 같으면 같은 플래그이므로 이름만으로 판정하게 바꿨고, 이 판정은 메모리를 읽지 않으므로 첫 주기부터 성립한다. 이전 판이 보고하던 432 는 6.4절의 가짜 리스트를 통해 한 주소에 아홉 번 쓴 것이 섞인 수다.

11. 논의

11.1 시작 시점 파일 방식과의 차이

실행 중에 쓴다. 값을 바꾸는 데 클라이언트 재시작이 필요하지 않다. 대신 서버 이동과 클라이언트 갱신이라는 무효화 조건을 구현이 직접 다뤄야 한다(9.2절).

결과로 센다. 설정 파일 방식에서 "적용됐다"는 파일에 썼다는 뜻이다. 여기서 적용은 메모리를 다시 읽어 요청한 값과 같았다는 뜻이며, 감사 경로는 쓰기 경로와 분리돼 있다. 넣지 못한 항목은 이름 단위로 사유가 남고, "이 빌드에 없다"는 판정은 구현 바깥의 증거로 검증한다(10.3절).

되돌릴 수 있다. 첫 쓰기 직전 값을 장부에 남기므로 되돌리기는 기본값 추측이 아니라 관측된 값의 복원이다. 장부는 프로세스 신원 네 가지가 모두 일치할 때만 재생된다(9.1절).

부수적으로 권한 요구가 작다. 원격 스레드·메모리 할당·페이지 보호 변경이 없고 접근 마스크는 `0x0438` 이다. 네트워크도 요구하지 않는다. 공개 설정표는 사용자가 명시적으로 새로 고칠 때만 가져오고 그 밖에는 캐시를 쓴다. 계정, 키 인증, 원격 갱신 확인, 분석 엔드포인트는 없다.

이름을 고르는 보조 기능(FlagML)은 이름 문자열을 문자 n-gram 해싱으로 256차원 희소 벡터로 만들고 (접두사 8개의 one-hot 포함), 은닉 64 유닛 ReLU, 출력 1 유닛 sigmoid 로 관련도를 매긴다. 학습은 온라인 SGD(학습률 0.02), 가중치는 사용자 프로필의 JSON 파일이다. 이 점수는 후보의 순위를 정할 뿐 쓰기를 승인하지 않는다.

주소·토큰·로그인·결제·제재·연령에 걸리는 이름 2,296개는 프로그램이 만드는 모든 목록에서 제외한다. 초기 구현은 부분 문자열로 대조해 `age` 가 `Percentage·Storage·Manager` 에, `ban` 이 `Bandwidth` 에 걸렸고, 그 시점 구현에서 6,387개가 말없이 제거됐다. 토큰 단위 대조로 바꾼 뒤 같은 시점의 제외는 1,711개였다.

11.2 한계

- 모든 수치는 한 대의 기기에서, 명시한 판의 한 클라이언트 빌드에 대해 측정했다. 다른 빌드에서 배치가 같다는 것은 보이지 않았다. 배치 상수는 실행 중 학습하지만 학습 자체의 일반성은 시험하지 않았다.
- 서버 이동 중의 동작은 재현하지 못했다. 9.2절의 장치는 그 순간의 신호를 합성 로그로 확인한 것이며, 실제 이동 중 크래시가 발생하지 않음을 보인 것이 아니다.
- 이름표(`+0xD0`)는 서술자의 3분의 1에서만 발견된다. 나머지에 대한 신원 확인은 vtable 검사에 의존하며, 이는 7.3절에 적은 대로 약한 증거다.
- 적용은 그 시점의 메모리 상태에 대한 진술이다. 서버가 같은 값을 다시 내려보내거나 클라이언트가 스스로 값을 바꾸면 그 자리는 더 이상 우리가 쓴 값이 아니다. 되돌리기는 그 경우를 구분해 보고하지만, 재적용 주기 사이의 상태까지 보장하지는 않는다.
- 크래시율(39세션에 1회)은 개발 중 축적된 관측이며 통제된 실험이 아니다.

12. 결론

실행 중인 클라이언트에 FastFlag 를 쓰는 경로를, 모든 쓰기 앞에 그것을 정당화하는 읽기를 두는 방식으로 구성했다. 구조 판정을 단일 시그니처가 아니라 독립 증거의 누적으로 바꾸고, 판정에 쓰는 배치 상수까지 실행 중 표본으로 배우게 하자, 빌드가 바뀌어도 같은 코드가 같은 규칙으로 판정한다. 성공을 쓰기 횟수가 아니라 다시 읽어 확인한 값으로 정의하자 두 가지가 가능해졌다. 넣지 못한 항목을 이름 단위로 설명할 수 있고, 그 설명을 구현 바깥의 증거로 검증할 수 있다.

302개 표에서 이 빌드가 가진 이름 75개 중 75개가 요청한 값을 들고 있었고, 넣지 못한 227개 중 구현의 책임으로 남은 것은 0개였다. 같은 경로에서 실행 후 첫 플래그까지의 지연은 약 8초에서 약 1.0초로 줄었으며, 그 과정에서 적용 결과는 이름 집합 수준에서 변하지 않았다.

부록 A. 재현 명령

명령	하는 일	본문
`--fflag-diag`	붙고, 배치를 채점하고, `flags.json` 을 파싱한다(쓰지 않음)	6
`--fflag-apply`	위와 같고, 이어서 주입한다	8
`--fflag-audit`	읽기 전용 감사: 쓴 것과 적용된 것을 분리	10.2
`--fflag-applied`	메모리를 다시 읽어 지금 적용 중인 부분집합을 JSON 으로 출력	10.2
`--fflag-why`	플래그마다 어느 단계에서 멈췄는지, 모듈 문자열표와 대조	3.5, 8.5
`--fflag-repair`	들어가지 않은 항목을 공개 설정표와 대조	10.3
`--fflag-locate <이름>`	이름 문자열 → `std::string` 헤더 → 서술자 노드를 직접 추적	3.1, 10.3
`--fflag-truth`	이 기기의 다른 목록들로 인덱스를 채점	10.3
`--fflag-ident`	이름표 오프셋을 인덱스 표본으로 채점	3.3
`--fflag-desc`	서술자 하나를 펼쳐 필드를 보여 준다	3.2
`--fflag-coverage`	이 빌드의 플래그 중 주입 가능한 비율	3.4, 10.4
`--fflag-regions`	영역 종류·크기·후보 수	5.4
`--fflag-scan-diff`	빠른 스윕과 전수 스윕을 이름 집합으로 비교	3.6, 10.6
`--fflag-time [n]`	한 번의 주입을 접속/인덱스/적용으로 분해	10.5
`--fflag-ready`	실행 후 첫 쓰기 가능 시점까지 측정	10.5
`--fflag-soak <주기 수> <초>`	재적용 루프를 반복해 적용 수의 범위를 본다	10.6
`--fflag-file-status`	시작 시점 파일이 무엇을 통과시켰는지 로그와 대조	2.3
`--fflag-safety`	자가 격리 장부의 상태	9.3
`--fflag-revert`	각 슬롯이 첫 쓰기 전에 들고 있던 값으로 복원	9.1
`--fflag-revert-status`	되돌리기 장부를 들여다본다(쓰지 않음)	9.1
`--fflag-verify <초>`	지정한 시간 뒤 값이 유지됐는지 다시 확인	—

표 8 — 본문의 각 수치를 다시 만드는 명령. 모두 창을 열지 않는다.

부록 B. 관측된 배치

기본값이며, 탐색이 갱신한다. 판정에 쓰는 값은 실행 중 학습한 값이다.

대상	오프셋	내용
싱글턴	`+0x08`	sentinel(빈 버킷 판정)
싱글턴	`+0x18`	버킷 배열 포인터
싱글턴	`+0x30`	마스크(= 버킷 수 − 1)
버킷	스트라이드 16, `+0x08`	체인 머리
엔트리	`+0x08`	다음 노드
엔트리	`+0x10`	이름(`std::string`)
엔트리	`+0x30`	서술자 포인터
서술자	`+0xC0`	값
서술자	`+0xD0` / `+0xD8`	이름 문자열 포인터 / 길이
`std::string`	`+0x00` / `+0x10` / `+0x18`	버퍼 union / size / capacity

표 9 — 이 빌드에서 관측한 배치.

가용성

본문의 모든 수치는 부록 A(표 8)의 명령으로 다시 만든다. 명령은 창을 열지 않고 stdout 과 `%APPDATA%\PandoraSettings\` 아래의 파일로 결과를 남기므로, 같은 기기에서 같은 절차를 반복할 수 있다. 엔진(L0–L5)은 프로세스 핸들에 의존하지 않아 `MemorySource` 를 모의 구현으로 바꾸면 Roblox 없이 단위 시험 160개가 그대로 돈다. 이 논문의 조판본(PDF)은 원고(`PAPER.md`)에서 `web/tools/paper.py` 로 생성한다.

측정에 쓴 클라이언트 빌드는 본문에 적은 시점의 것이고, 배치 상수는 실행 중 학습하므로 다른 빌드에서 같은 값이 나온다고 보장하지 않는다.

감사의 글

대조용 목록은 공개된 부트스트래퍼와 커뮤니티 목록을 관리하는 오픈소스 기여자들에게서 얻었다 (10.3절의 채점에 쓴 Froststrap 목록 40개 포함). 인덱스 밖의 증거로 쓸 수 있었던 것은 그 목록들이 공개돼 있었기 때문이다.

참고문헌

1. Microsoft, *OpenProcess*, *ReadProcessMemory*, *WriteProcessMemory*, *VirtualQueryEx* — Windows API 문서.
2. Pandora 기술 문서 PANDORA-TECH-001, v1.1.0 (2026-09).
3. 본 저장소의 `src/fflag_engine.rs`(엔진 L0–L5), `src/fflag_injector.rs`(실행 경로), `src/memory.rs` (핸들), 그리고 단위 시험 160개.