Attacker Control and Bug Prioritization

Guilhem Lacombe
Université Paris-Saclay, CEA, List, France
guilhem.lacombe.97@gmail.com

Sébastien Bardin
Université Paris-Saclay, CEA, List, France
sebastien.bardin@cea.fr

Abstract

As bug-finding methods improve, bug-fixing capabilities are exceeded, resulting in an accumulation of potential vulnerabilities. There is thus a need for efficient and precise bug prioritization based on exploitability. In this work, we explore the notion of control of an attacker over a vulnerability's parameters, which is an often overlooked factor of exploitability. We show that taint as well as straightforward qualitative and quantitative notions of control are not enough to effectively differentiate vulnerabilities. Instead, we propose to focus analysis on feasible value sets, which we call *domains of control*, in order to better take into account threat models and expert insight. Our new *Shrink and Split* algorithm efficiently extracts domains of control from path constraints obtained with symbolic execution and renders them in an easily processed, human-readable form. This in turn allows to automatically compute more complex control metrics, such as *weighted Quantitative Control*, which factors in the varying threat levels of different values. Experiments show that our method is both efficient and precise. In particular, it is the only one able to distinguish between vulnerabilities such as *cve-2019-14192* and *cve-2022-30552*, while revealing a mistake in the human evaluation of *cve-2022-30790*. The high degree of automation of our tool also brings us closer to a fully-automated evaluation pipeline.

1 Introduction

Over the past decades, program analysis research has predominantly focused on expanding and refining bug-finding methods. As a result, automated techniques such as fuzzing, symbolic execution and data-flow analysis have improved significantly. Fuzzing in particular is very prolific nowadays, with efforts such as Syzbot [1, 2] uncovering thousands of bugs. In addition, the popularization of open-source development models led to even greater bug-finding capabilities as third parties are able to contribute additional man- and processing-power.

The problem. Unfortunately, bug-fixing capabilities have not seen the same level of development and thus unfixed bugs tend to accumulate over time. This tendency is especially striking when looking at Syzbot's bug reports for the Linux kernel: 1561 open bugs as of December 2024, the oldest being over seven years old. Furthermore, there are 51 mentions of "out-of-bounds" and 120 of "use-after-free". Fuzzing of smaller scale projects as performed by Google's OSS-fuzz [3] also discovers large amounts of bugs, with over 1500 labeled "vulnerability" in 2024 alone, including 427 buffer overflows, all of uncertain security impact.

This poses a severe security risk as overwhelmed developers are unable to identify true vulnerabilities and thus dispatch adequate bug-fixing efforts. Thus a need for precise yet efficient bug prioritization arises.

Due to the diversity of requirements and challenges inherent to different development projects, the choice of criteria for bug prioritization is ultimately up to the developers. For example, bugs causing infinite loops may be of higher priority in a server than in an offline application due to the greater focus on availability. However this choice is limited by the capabilities of current analyzers and error detectors, which typically focus on detecting *classes* of vulnerabilities, such as use-after-free and out-of-bounds memory accesses in the case of the Linux kernel sanitizer KASAN [4]. For this reason, current bug prioritization practices often amount to simply attributing varying threat levels to different classes of vulnerabilities and are thus very coarse-grained.

In order to establish finer-grained prioritization criteria, one may consider using Automated Exploit Generation (AEG) [8, 34] to automatically check for exploitability. Ignoring the technical challenges that would be involved, there are still major issues with this approach, such as the fact that AEG gives no guarantees if no exploits are found. Indeed, a lack of generated exploit only indicates incompatibility with the AEG method. There is also a fundamental divergence in objectives: the goal of AEG is first and above all to build functional exploits, not necessarily reusing the same exact bug given as input. Most AEG engines only use input bugs as starting points, some even only use them as seeds in their own bug search [16, 66, 69].

On the other hand, there are lessons to be learned from the inner workings or AEG engines. In particular, many have some variant of a rating system for vulnerabilities in order to select the most adequate ones for the exploit being built. Typically, vulnerabilities offering some level of control are preferred. For example, KOOBE [16] compares out-of-bounds write vulnerabilities based on obtainable offsets, sizes and written values, and prioritizes those with a wider range of capabilities. However its approach only allows for very limited partial ordering and is thus inadequate for generic bug prioritization. *An attacker's control over the value of vulnerability parameters is thus an important factor of exploitability.*

Our goal. We aim to develop fine-grained bug prioritization methods based on automatically estimating *attacker control* over vulnerability parameters.

While attacker control over a variable or expression is generally understood to refer to the ability of an attacker to influence its value through program inputs, the lack of formal definitions renders this concept ambiguous. Taint and symbolic bytes are often implicitly considered as qualitative indicators of control during taint analysis and symbolic execution respectively [34, 57, 69], despite a lack of formal guarantees. On the other hand, Newsome et al. [53] define a quantitative notion of control related to channel capacity, which assumes all values are equally dangerous and can thus be misleading. For example, very large write sizes in a buffer overflow write will likely result in a crash, while smaller ones are more likely to hit critical data such as pointer or return addresses. Furthermore, applicable algorithms scale poorly and the authors did not explore applications to bug prioritization. *Overall, attacker control lacks a unified theoretical framework as well as reasonably scalable approaches able to capture nuances such as the varying threat levels of different values.*

Our proposal. We propose to focus control analysis on the *domains of control*, i.e., the sets of feasible values for vulnerability parameters. Our approach consists in deriving the path constraint corresponding to program execution on a known vulnerability-triggering input using single-path symbolic execution, then use our *Shrink and Split* algorithm to measure domains of control for a variable or expression in an analysis-friendly form. In particular, the absence of path exploration eliminates the risk of path explosion, bringing our symbolic execution more in line with dynamic taint analysis [57] in terms of scalability. Finally, vulnerabilities can be scored with control metrics based on these domains of control, such as *weighted Quantitative Control*, the sum of the threat levels of the different feasible values according to some weight function. For example, we can compute a score for out-of-bounds write vulnerabilities by combining the weighted quantitative control over the write offset, size and data.

While this work mainly focuses on vulnerabilities involving well defined data-flows, in particular out-of-bound memory bugs, we discuss applicability to other vulnerabilities as well.

Contributions. We claim the following contributions:

- We provide a generic, unified and analysis-method-agnostic theoretical framework for the notion of attacker control over the value of a given expression. We define the *domain of control* as the set of obtainable values (Section 5.1). Then we define *weak control* as the ability of an attacker to change the value through inputs and *strong control* as the existence of inputs leading to every possible value (Section 5.2). Finally, we define *quantitative control* (Section 5.3) based on existing notions and expand it into *weighted quantitative control* (Section 5.4). We then argue that *taint analysis* can only ensure the absence of weak control (Section 6.1), strongly limiting its usefulness for control analysis;
- We propose *Shrink and Split* (Section 6.3), a novel algorithm able to efficiently determine domains of control as a set of intervals with weak and strong control guarantees. It extracts them from logical formulas representing path constraints, which can be obtained with standard techniques such as symbolic execution or bounded model checking. Shrink and Split is a refinement process entirely based on qualitative analysis and yields approximate results when interrupted. Additional regularity constraints, such as fixed bits, can also be taken into account to improve precision. We also provide algorithms for measuring weak, strong and quantitative control;
- We implemented Shrink and Split and our other algorithms into a new dynamic analysis tool with symbolic execution capabilities. Measuring control with Shrink and Split is significantly more precise than using taint analysis. Our algorithm also performs significantly better than the closest one from the literature (Newsome et al. [53], see Section 7). Furthermore, Shrink and Split is more robust and more informative than counting algorithms. Our tool also implements various utilities enabling automatic end-to-end analysis;
- We perform a ground-truth evaluation on real-world vulnerabilities (Section 7.6) which shows that our approach gives a more precise exploitability assessment than CVSS scores and quantitative information flow analysis. In particular, we are able to clearly differentiate vulnerability capabilities, match identical ones and even help to correct previous human evaluation in the case of *cve-2022-30790*. The high degree of automation of our tool also allowed us to evaluate memory out-of-bounds bugs from the Magma fuzzing benchmark [33] in an end-to-end manner and realistic field conditions (Section 7.7). We were additionally able to show that the results from this experiment match the tendencies of coarse-grained human expectations.

This work constitutes a step toward precise, fully-automated exploitability assessment. While exploitability as a concept is difficult to formally grasp and may not be reducible to a single catch-all notion, control is definitively an interesting part of it. Combinations with other indicators is an interesting research direction.

2 Motivating Example

```
1  #define HEADER_SIZE 40
2
3  uint64_t check_header(char *input,
4          uint64_t input_size)
5  {
6      //2) input[0->7] written on the stack
7      uint64_t header = *((uint64_t *) input);
8      return header <= 296;
9  }
10
11 void get_msg(char *buf, uint64_t buf_size,
12         char *input, uint64_t input_size)
13 {
14     //3) not initialized => size = header
15     uint64_t size;
16     if(input_size <= buf_size + HEADER_SIZE)
17         //4) input_size < 40 => integer underflow
18         size = input_size - HEADER_SIZE;
19     //5) buffer overflow!!!
20     // a. input_size < 40
21     //    => 2^64 - 40 <= size < 2^64
22     // b. input_size > 296 => size = header
23     memcpy(buf, input + HEADER_SIZE, size);
24 }
25
26 int main(...)
27 {
28     //1) inputs: char *input, uint64_t input_size
29     ...
30     char buf[256];
31     if(check_header(input, input_size))
32         get_msg(buf, 256, input, input_size);
33     ...
34 }
```

Listing 1: Motivating example

In order to illustrate the problem of bug prioritization, consider the program from Listing 1, which exhibits two similar vulnerabilities with different levels of exploitability.

Explanation. We assume that *input_size* and *input* are independently provided by the attacker, thus *input_size* is not constrained by *input*. However the program assumes that it is and that *input* has a header of size *HEADER_SIZE* that must be removed. This leads to unintended behaviour when passing malformed inputs.

Vulnerability a is triggered when *input_size* is smaller than *HEADER_SIZE*. In this scenario, the branch condition on line 16 is true, leading to an integer underflow on line 18. A buffer overflow then occurs at line 23 with $2^{64} - 40 \leq size < 2^{64}$, the size of *buf* being only 256.

Vulnerability b occurs when *input_size* is greater than $buf\_size + HEADER\_SIZE$. This time, the branch condition on line 16 is false and *size* remains uninitialized. Its value is therefore equal to the first 8 bytes of *input* which were written on the stack on line 7. The attacker can thus obtain $257 \leq size \leq 296$ on line 23 and cause a buffer overflow.

Discussion. While the main impact of both vulnerabilities is a buffer overflow allowing an attacker to overwrite the stack, their levels of exploitability differ greatly. Indeed vulnerability *a* always results in a crash as the program attempts to overwrite roughly 2^{64} bytes of memory. On the other hand, vulnerability *b* allows an attacker to overwrite at most 40 bytes of memory with data they provide. This level of *control* makes vulnerability *b* easier to exploit. For example, the attacker can overwrite the return address of the *main* function and hijack control-flow.

In the context of security-aware bug prioritization, vulnerability *b* should be fixed before vulnerability *a*. However it would be difficult to recognize this fact in practice when only vulnerability types are reported. In this case, information about how *controllable* vulnerabilities *a* and *b* are is required to give them the correct level of priority. Our experimental evaluation shows that this scenario can indeed happen in practice (see Table 5 in Section 7).

Prior work. Commonly used indicators of control such as taint and symbolic bytes [57, 69] are not helpful here: *size* would be tainted and symbolic on line 23. It also can take the same number of values in both vulnerabilities thus quantitative information flow analysis [46] is equally unhelpful. Correctly handling such cases requires a new approach, which we detail in Sections 5 and 6.

3 Background

Let P be a program. An execution of P is a sequence of states each associated with a location corresponding to the current instruction. Let S_P be the set of possible states of P, L_P the set of locations of P and $\lambda(s) \in L_P$ the program location associated with a state s. We note $\rightarrow: S_P \times S_P$ the transition relation between states during execution of P and $\rightarrow^*$ its transitive closure.

Sequences of states are called *traces* and sequences of locations *paths*. A path $\pi = (l_i)_i$ is said to be feasible *iff* there exists a possible trace $t = (s_i)_i$ of P following π, i.e., forall i, $\lambda(s_i) = l_i$. If π starts with s and ends with s', we note $s \rightarrow_\pi s'$.

Let V_P be the set of variables of P and I_P its set of possible inputs. We note $s(v)$ the value of a variable v in a state s and $Dom(v)$ the set of values that v can take a priori (e.g., considering its type). To simplify notations, *we assimilate expressions over variables as implicit variables.* For any input $i \in I_P$ and state $s \in S_P$, we note $i \rightarrow^* s$ *iff* $s_0 \rightarrow^* s$ with s_0 the initial state corresponding to input i.

3.1 Taint Analysis

Taint analysis [57] refers to the static or dynamic propagation of tags associated to data to qualitatively track data-flows. An expression introducing a tag is referred to as a *source* while program locations where the presence of tags is checked are called *sinks*. This technique has the advantage of being fairly lightweight at the cost of limited precision and guarantees. A common application is the verification and enforcement of non-interference properties [31]. Some prior works use taint analysis to detect attacker control [34, 45].

3.2 Symbolic Execution

Symbolic Execution (SE) [57] refers to the exploration of execution paths in a program while computing the associated constraints on inputs and other variables such as registers and memory locations. These constraints usually take the form of a Satifiability Modulo Theory (SMT) formula and are used to check whether paths are feasible. Variables dependant on inputs are referred to as *symbolic* variables while constant values are said to be *concrete*. While SE excels at covering edge-case paths and allows to generate inputs triggering them, it may suffer from *path explosion* , i.e., an exponential increase in the number of paths being followed. Some constraints may also be too difficult to solve (typically, cryptography). SE engines include KLEE [13] at LLVM-IR level as well as Angr [61] and BINSEC [21, 24] for binary analysis.

Choice of SMT theory. We assume that symbolic states are expressed in the *ABV* theory (arrays and bitvectors), with variables represented as bitvectors and memory as a logical array. As a consequence, the values of variables are *finite*.

Notations. We note $SE(\pi)$ the symbolic state obtained at the end of the symbolic execution of a path π, with program inputs initially symbolic. Let ϕ be a symbolic state. We define the following notations:

- $\phi(x) \triangleq True$ if x satisfies ϕ, $False$ otherwise
- $sat(\phi) \triangleq True$ if $\exists\, x : \phi(x)$, $False$ otherwise
- $val(\phi, v)$ a feasible value of v in ϕ, nil if none exists
- $val(\phi(x), v)$ the value of v for the input x if $\phi(x)$, else nil
- $duplicate(\phi, v) \triangleq \phi', v'$ with ϕ' a copy of ϕ with separate variables, v' being equivalent to v

3.3 Quantitative Information Flow

Quantitative information flow analysis consists in evaluating the quantity of information flowing through a channel, often by measuring channel capacity [23]. This is usually done to quantify the leakage of secret information [18, 37, 51].

Projected Model Counting (PMC). While model counting counts the number of solutions of a formula, PMC counts the number of values a subset of free variables can take within a SAT formula [10]. It can be used to measure QIF by converting SMT formulas generated with symbolic execution or model checking into SAT formulas [11, 46]. We note SE_{PMC} the algorithm combining SE and PMC.

While the precision of current exact PMC solvers such as D4 [47] and Ganak [60] is high, scalability is an issue and they may even fail on trivial examples as shown in Section 7. ApproxMC [15, 62, 63] is an efficient approximated PMC solver with probably approximately correct guarantees, i.e., results are within a given error margin with a given probability.

Quantitative Influence. Newsome et al. [53] defined the notion of quantitative influence, derived from channel capacity, as the logarithm of the number of feasible values for a variable. We integrate this notion into our control framework as *quantitative control* in Section 5.3. They also proposed an algorithm for measuring quantitative influence based on statistical sampling, which we discuss in Section 6.3 and evaluate against in Section 7.

4 Problem Statement

Our goal is to rank vulnerabilities according to how controllable their parameters are, with as little assumptions on the goal of a potential attacker as possible, and as little human effort as possible.

4.1 Requirements

Vulnerability information. We assume that the *type of vulnerability* (e.g., out-of-bounds write, use-after-free, pointer corruption, etc.) is known. We also assume that we have a *triggering input*. These parameters can be expected to be available for most discovered vulnerabilities as proof of existence and for debugging purposes. Our method can also be paired with automated bug finding methods such as symbolic execution [57] or fuzzing [50] to directly find inputs.

Human expertise. Identifying vulnerability parameters, which we refer to as *target variables*, may require some human expertise, although they can be identified automatically in most cases as shown in Section 7.7. These parameters vary depending on vulnerability types: size and data of a buffer overflow write, value of a corrupted pointer, etc. They can be implicit, meaning that the value itself is not stored anywhere at runtime, such as the overall write size from a *strcpy* call.

4.2 Archetypal Control Problems

Notions of control can relate to a wide array of vulnerabilities in many ways. Rather than considering individual types of vulnerabilities, we define archetypal control problems:

Data control problem. This problem is the most straight forward: the vulnerability allows an attacker to control the value of a well-defined structure, such as a pointer or a used-after-free object. In this case, we only need to analyze the degree of control of the attacker over the value of this structure.

This archetype fits vulnerabilities such as some use-after-frees, use-before-initialization and pointer corruption.

Memory range control problem. This problem covers vulnerabilities allowing an attacker to read or write variable size data at variable memory addresses, without any explicit structure. Here the vulnerability's parameters are the address, size and written data when applicable. As variable-size data can be difficult to handle, one may resort to extrapolating from fixed-size analysis.

This archetype fits most out-of-bounds read / write vulnerabilities, such as vulnerabilities *a* and *b* from Listing 1.

What about vulnerabilities without explicit data-flows? Control is inherently tied to data-flows. However, the effect of those data-flows are not always materialized into an explicit value. For example, attackers may control the number of memory allocator interactions in order to achieve heap layout manipulation [35, 36, 67]. In such cases, parameters must be made explicit at the code level or within the analysis, for example with counters.

Scope. *We choose to mostly focus on memory range problems in this work* (e.g., buffer overflows) to avoid scope creep and since they match the typical bugs found by fuzzers. Our benchmark nevertheless also contains examples of the data problem (e.g., pointer corruption after a use-after-free).

5 Formally Defining Control

In order to explicitly define various aspects of the notion of control, we first need to precisely identify the meaning of *controlling a variable* from a natural language standpoint. We consider attackers interacting with a program through its inputs, with the goal to take advantage of a given vulnerability (e.g., buffer overflow) with several variable parameters (e.g., the size of a buffer overflow write), for some malicious purpose (e.g., rewriting a return address or a function pointer).

The *weakest level of control* attackers are looking for is the ability to *influence* the value of the vulnerability parameters through inputs, i.e., to obtain different values for them. In other words, attackers are not satisfied with simply knowing of the existence of a data-flow from inputs to the targeted (parameter) variables, it must also be influenceable. On the other hand, the *strongest level of control* the attacker is seeking is the ability to obtain any value for the vulnerability parameters.

We propose hereafter a hierarchy of notions of control with precise formal definitions. All will be relevant in Section 6 when we propose algorithms for control evaluation.

5.1 Domain of Control

First let us define our key notion of *Domain of Control*, from which we derive all our other notions of control.

Given a program *P*, we define the *Domain of Control* of variable v at location l as the set of feasible values for v at this location, i.e., the set of values e of v for which we can find some input i such that executing *P* on i leads to a program state s (at location l) in which v evaluates to e. More formally, we have the following definition:

Definition 1 (Domain of Control)**.**

$$DoC(v,l) \triangleq \{e \in Dom(v) / \exists i \in I_P : i \to^* s \in S_P$$

$$\textit{with } \lambda(s) = l,\ s(v) = e\}$$

5.2 Qualitative Control

We now define *Weak Control* – resp. *Strong Control* – as the attacker's ability to find inputs of the program leading to different values – resp. leading to any value – of v at l.

Definition 2 (Weak Control (WC))**.** *Given a program P, a variable $v \in V_P$ is weakly controlled at location $l \in L_P$ if there exists $i, i' \in I_P$ two inputs such that $i \to^* s$ and $i' \to^* s'$ with $s, s' \in S_P$, $s(v) \neq s'(v)$ and $\lambda(s) = \lambda(s') = l$. We note $WC(v,l)$.*

Note that v is weakly controlled at l *iff* $|DoC(v,l)| > 1$, hence *iff* v can indeed take at least two values, depending on the input chosen by the attacker.

Definition 3 (Strong Control (SC))**.** *We say that (v,l) is strongly controlled if for all $e \in Dom(v)$, there exists an input $i \in I_P$ such that $i \to^* s \in S_P$ with $s(v) = e$ and $\lambda(s) = l$. We note $SC(v,l)$.*

In this case, v is strongly controlled at l *iff* $|DoC(v,l)| = |Dom(v)|$, hence *iff* v can take any value from its definition domain. Intuitively, *SC* is a stronger property than *WC*.

Proposition 1. *Strong control is stronger that Weak control, i.e., for any program P and target (l,v), $SC(v,l) \Rightarrow WC(v,l)$.*

```
int x = input;
//here x is strongly controlled
if(x)
    //here x is weakly but not strongly controlled
if(!x)
    //here x is neither weakly nor strongly
    //controlled
```

Listing 2: Examples of Weak and Strong Control

The main drawback of these definitions is a lack of nuance, as weak and strong control guarantee the lowest and highest amount of control possible respectively as shown on Listing 2. In the case of our motivating example (Listing 1), the out-of-bounds write size is weakly but not strongly controlled for both vulnerabilities, thus these notions cannot be used to distinguish between them. Nevertheless, we show in Section 6.3 that *WC* and *SC* can be used as formal guarantees within more subtle approaches.

5.3 Quantitative Control

Shifting to quantitative measurement is a common approach when attempting to overcome the limits of qualitative analysis [9, 26, 32, 37]. Following this trend and similarly to Newsome et al. [53], we define *Quantitative Control* as the (normalized) channel capacity [23] of v.

Definition 4 (Quantitative Control (QC)). *(similar to Newsome et al. [53])*

$$QC(v,l) \triangleq \frac{ln(|DoC(v,l)|)}{ln(|Dom(v)|)}$$

The value of QC relates intuitively to WC and SC.

Proposition 2. $WC(v,l) \iff QC(v,l) > 0$

Proposition 3. $SC(v,l) \iff QC(v,l) = 1$

While QC allows for a level of precision well beyond WC and SC, measuring it in practice is challenging due to the poor scalability of counting algorithms (see Section 7). In addition, QC remains a one-dimensional measure of control: it assumes that all possible values are equally dangerous. However, this is often not the case as in our motivating example (Listing 1), where both vulnerabilities have the same number of write sizes but the larger write sizes of *vulnerability a* lead to a crash and thus have a lower threat level.

5.4 Weighted Quantitative Control

In light of the limitations of WC, SC and QC, we argue that fine-grained evaluation of attacker control should focus on the domains of control and account for expert insight into which values are more dangerous.

For instance, in our motivating example (Listing 1), we see that buffer overflow write sizes for *vulnerability a* are too large to be useful as they will lead to crashes, while those for *vulnerability b* allow to overwrite nearby stack data only and are therefore more useful for the attacker. Generally, smaller overflow write sizes / offsets should be worth more due to the lower variability of near targets and their potential interest for the attacker (chunk metadata, adjacent objects, return addresses and other stack data, etc.).

One way to automatize such reasoning is to compute a weighted quantitative control metric.

Definition 5 (Weighted Quantitative Control (wQC)). *Let $\omega_v : Dom(v) \to \mathbb{R}$ be a weight function.*

$$wQC(v,l,\omega) \triangleq \frac{\sum_{n \in DoC(v,l)} \omega(n)}{\sum_{n \in Dom(v)} \omega(n)}$$

This may be difficult to compute in practice, however in some instances it is possible to efficiently approximate.

Proposition 4. *Let $Dom(v) = [\![a,b]\!] \subset \mathbb{N}$, P be a set of intervals in $\mathbb{N}$ and a partition of $DoC(v,l)$. Let $\Omega : \mathbb{R} \to \mathbb{R}$ be an integrable function on $[a,b]$ and $\omega = \Omega|_{Dom(v)}$.*

$$wQC(v,l,\omega) \approx \frac{\sum_{[\![i,j]\!] \in P} \int_i^{j+1} \Omega(x)dx}{\int_a^{b+1} \Omega(x)dx} \tag{1}$$

If we have additional constraints on intervals[1]*, with $\rho(I)$ the actual number of feasible values in $I \in P$, we get:*

$$wQC(v,l,\omega) \approx \frac{\sum_{[\![i,j]\!] \in P} \frac{\rho([\![i,j]\!])}{j-i} \int_i^{j+1} \Omega(x)dx}{\int_a^{b+1} \Omega(x)dx} \tag{2}$$

For example, for buffer overflow / underflow write vulnerabilities such as in our motivating example, we can use $\omega : x \mapsto \frac{1}{ln(2)x}$ for out-of-bounds write sizes and offsets, in order to introduce bias toward smaller values, i.e., tampering of local data. This way, the weighted quantitative control value for the overflow size, i.e., the write size minus the size of the buffer, is approximately $\frac{\int_{2^{64}-296}^{2^{64}-256} \omega(x)dx}{\int_1^{2^{64}} \omega(x)dx} = \frac{log_2(2^{64}-256) - log_2(2^{64}-296)}{log_2(2^{64})} \approx 0$ for *vulnerability a* and $\frac{\int_1^{41} \omega(x)dx}{\int_1^{2^{64}} \omega(x)dx} = \frac{log_2(41)}{log_2(2^{64})} \approx 0.0837$ for *vulnerability b*, thus giving a clear order of priority.

Exact, constraint-based domain of control representations, such as SMT formulas, are often too complex to perform such analysis. We present a solution to provide a simplified representation of $DoC(v,l)$ in Section 6.

5.5 Variants

Control under assumption. So far, we have defined control over the entirety of $Dom(v)$. However it can be useful to verify control properties over a subset $E \subset Dom(v)$ of feasible values, as it would allow to factor in assumptions such as v being always even or within a given range. We thus define notions of control over a subset of values similarly to their regular counterparts with $Dom(v)$ reduced to E, denoted with the suffix $|_E$. In practice, we make use of the notion of strong control under assumption in Algorithm 4 in Section 6.3.

Restriction to a single path. Reasoning over a single path is advantageous in the context of program analysis as it eliminates loops. We define DoC_π, WC_π, SC_π, QC_π and wQC_π similarly to their non-single-path counterpart, with the constraint that values are obtainable through a single path π, i.e., $\to^*$ is replaced with $\to_\pi$.

While this approach comes at the cost of completeness, many vulnerabilities can still be adequately analyzed this way. Furthermore, domains of control for different paths can be merged without loss of precision. The limits of this approach and potential solutions are further discussed in Section 8.

[1] For example, $x \equiv 0[2]$

5.6 Conclusion

To sum up, weak and strong control are too extreme to be insightful on their own. Quantitative control allows for more precision, however it still remains uni-dimensional. This is an issue since control is a complex, multi-dimensional notion with different values potentially having different threat levels. We propose to focus more on this aspect by directly analyzing domains of control and distilling them down with more expressive metrics, such as weighted quantitative control.

6 Evaluating Control

Assuming vulnerability-triggering inputs are available, we choose to employ dynamic analysis methods at binary level in order to precisely follow complex program behaviour at the cost of restricting analysis to a single execution path π.

We will first discuss why taint is a poor indicator of control, before detailing symbolic-execution-based solutions.

6.1 Taint Analysis Cannot Guarantee Control

A typical taint policy consists in propagating taint to the outputs of an instruction if at least one of its inputs is tainted (see Appendix I for an example). While one may be tempted to interpret taint as an indicator of control, it unfortunately cannot give much guarantees.

Proposition 5 (Taint is limited for control evaluation)**.** *Taint only guarantees that* $\neg tainted \Rightarrow \neg WC$.

```
1 int x = input; ← tainted
2 int y = input;
3 int z = x + y;
4 if(x >= 0)
5 {
6     if(x <= 0) ← x tainted but can only be equal to 0
7 }
8 int w = z - x; ← w tainted but can only be equal to y
```

Listing 3: Weak Control false positives with taint analysis

If a variable is tainted, there is no guarantee that its value can change depending on inputs, i.e., that it is weakly controlled. While taint can be removed when a single instruction restricts a variable to a single value (e.g., $if(!x)$), this can also happen due to arbitrarily complex constraints imposed by multiple instructions (lines 6 and 8 on Listing 3). Handling those cases with precision thus requires some form of constraint tracking, i.e., symbolic execution or other similar techniques. This issue is reflected in the literature with works using lightweight symbolic execution to refine dynamic taint propagation [52,65].

In conclusion, taint analysis cannot prove weak control, the lowest level of control in our framework. It is thus insufficient for evaluating control with any degree of precision.

6.2 Verifying Weak and Strong Control with SMT Solvers

As discussed previously, we need to analyze path constraints in order to properly evaluate control, hence our reliance on symbolic execution. The following algorithms check Weak and Strong Control for a variable v in a symbolic state ϕ.

Algorithm 1 SE_{WC}

```
Require: $v$ a variable, $l$ a location, $\pi$ a path ending at $l$
Ensure: returns $true \iff WC_\pi(v,l)$
  $\phi \leftarrow SE(\pi)$
  $\phi', v' \leftarrow duplicate(\phi, v)$
  return $sat(\phi \wedge \phi' \wedge v \neq v')$
```

Algorithm 1 checks WC by checking satisfiability for two different values of v. It is correct and complete for WC_π.

Proposition 6. $SE_{WC}(v,l,\pi) \iff WC_\pi(v,l)$

Algorithm 2 SE_{SC}

```
Require: $v$ a variable, $l$ a location, $\pi$ a path ending at $l$
Ensure: returns $true \iff SC_\pi(v,l)$
  $\phi \leftarrow SE(\pi)$
  procedure SC($v,l,\phi,E$)                          ▷ $E \subseteq Dom(v)$
    $\phi' \leftarrow \exists\, y \in E\colon \forall\, x,\ \phi(x) \Rightarrow val(\phi(x), v) \neq y$
    return $sat(\phi')$, $val(\phi', y)$
  end procedure
  $res, y \leftarrow$ SC($v,l,\phi,Dom(v)$)          ▷ $y$: counterex. if not $nil$
  return $\neg res$
```

Algorithm 2 checks SC by searching for a counterexample, i.e., an infeasible value. It is correct and complete for SC_π.

Proposition 7. $SE_{SC}(v,l,\pi) \iff SC_\pi(v,l)$

Algorithm 3 $SE_{SC}|_E$

```
Require: $v$ a variable, $l$ a location, $\pi$ a path ending at $l$, $E \subseteq Dom(v)$
Ensure: returns $true \iff SC_\pi|_E(v,l)$
  $\phi \leftarrow SE(\pi)$
  $res, y \leftarrow$ SC($v,l,\phi,E$)          ▷ $y$: counterexample if not $nil$
  return $\neg res$
```

Similarly, Algorithm 3 checks SC over $E \subseteq Dom(v)$ ($SC|_E$) by limiting the search for an infeasible value to E.

Solver requirements. While SE_{WC} can be verified by any SMT solver, SE_{SC} requires quantifier support. In addition, obtaining the counterexample y requires the ability to extract models in quantified SMT formulas.

6.3 Extracting Domains of Control with Shrink and Split

Giving a simple characterization of $DoC_\pi(v,l)$ as a set of intervals would greatly help further analysis, for example allowing to compute weighted quantitative control as discussed in Section 5. To achieve this, we propose the *Shrink and Split* approach detailed in Algorithm 4, which consists in repeatedly *shrinking* and *splitting* $Dom(v)$ until we reach $DoC_\pi(v,l)$.

Algorithm 4 $SE_{S\&S}$

```
Require: $v$ a variable, $l$ a location, $\pi$ a path ending at $l$
Ensure: returns $DoC_\pi(v,l)$
  $\phi \leftarrow SE(\pi)$
  procedure S&S($v, l, \phi, i, c$)          ▷ $c$: additional constraints
                        - Shrinking -
    $\phi \leftarrow \phi \wedge v \in i$
    $lo \leftarrow min(val(\phi, v))$
    $hi \leftarrow max(val(\phi, v))$
    $i \leftarrow [lo; hi]$
    $\phi \leftarrow \phi \wedge v \in i$
                   - Checking for Strong Control -
    $sc, y \leftarrow$ SC($v, l, \phi, \{y \in i / c(y)\}$)
    if $sc$ then
                      - No need to split -
        return $i$
    else
                     - Splitting around $y$ -
        return S&S($v, l, \phi, [lo; y[, c$)
                  $\cup$ S&S($v, l, \phi, ]y; hi], c$)
    end if
  end procedure
  return S&S($v, l, \phi, Dom(v), True$)
```

Shrinking. Intervals are *shrunk* by finding their *feasible bounds*. To achieve this, we need a *min* and a *max* directive. In practice we use the SMT solver Z3's optimization modules, which rely on MaxSMT solvers in the case of bitvectors [12]. Alternatively, feasible bounds can be determined using a binary-search-like method, requiring $O(log(|Dom(v)|))$ calls to the solver. As a result, infeasible values outside of those feasible bounds are eliminated.

Checking for Strong Control. We then check whether v is strongly controlled over the shrunk interval i. An affirmative result indicates that i is a subset of $DoC(v,l)$, otherwise we know of at least one infeasible value y.

Splitting. If infeasible values remain in i, we *split* it around y and repeat the *S&S* process on both halves.

Since variables are represented as finite bitvectors, all infeasible values are eventually eliminated and we are left with a union of intervals exactly matching $DoC(v,l)$.

Proposition 8. $DoC_\pi(v,l) = SE_{S\&S}(v,l,Dom(v),\pi)$

Practical Limits. In practice, we observe that proving strong control or finding a counterexample can fail, due to the complexity of constraints, time-outs, solver limitations or bugs... In this case, weak control at least is guaranteed on the current interval as feasible bounds were found. In the worst case, *S&S* may only be able to show weak control over the entire domain, although this never happens in our experiments.

In addition, *S&S* has an obvious flaw: when the domains of control contains a lot of holes, the number of splits required may be impractically large.

Mitigation 1: limited splitting. One way to avoid excessive splitting is to set a limit and interrupt *S&S* when it is reached. This can be a good solution as intermediate results get more precise over time, although it sacrifices exactness.

Mitigation 2: fixed bits. Another possible mitigation is to identify fixed bits and take them into account with an additional constraint, preventing splitting caused by them.

Algorithm 5 $SE_{S\&SFB}$: $SE_{S\&S}$ with a fixed bits constraint

```
Require: $v$ a variable, $l$ a location, $\pi$ a path ending at $l$
Ensure: returns $DoC_\pi(v,l)$
  $\phi \leftarrow SE(\pi)$
  $\phi', v' \leftarrow duplicate(\phi, v)$
  $\phi'', v'' \leftarrow duplicate(\phi, v)$
  $\phi'' \leftarrow \phi \wedge \phi' \wedge \phi'' \wedge mask = \sim (v' \hat{} v''))$
      $\wedge bits = v' \& v''$                              ▷ (a)
  $\phi'' \leftarrow \phi'' \wedge (\neg \exists x\colon \phi''(x)$
      $\wedge val(\phi''(x), v) \hat{} mask \neq bits))$              ▷ (b)
  if $sat(\phi'')$ then
      $mask \leftarrow val(\phi'', mask)$
      $bits \leftarrow val(\phi'', bits)$
  else
      $mask \leftarrow 0$
      $bits \leftarrow 0$
  end if
  return S&S($v, l, \phi, Dom(v), y \mapsto y \hat{} mask = bits$)
```

Algorithm 5 shows how we compute fixed bits and incorporate them into *S&S*. (a) and (b) ensure that *mask* corresponds to at least and at most all fixed bits respectively, while *bits* contains their value.

There exist corner cases where two feasible values always have more common bits than the number of overall fixed bits, in which case this algorithm fails with *mask* and *bits* null. However we expect such cases to be rare. Alternatively we could evaluate each bit individually, however we observe that our approach is more efficient in practice.

Once the fixed bits constraint is determined, we incorporate it within *S&S* by checking strong control over values satisfying it in the shrunk interval. This implicitly eliminates holes and prevents excessive splitting.

This variant of *S&S* is also evaluated in Section 7 and shown to improve precision with no noticeable overhead.

Other possible mitigations. Other types of constraints such as congruences or polyhedra could be similarly considered in lieu of fixed bits. This is left as future investigation.

Approximation. When an exact results cannot be computed, Shrink and Split still gives an over- and under-approximation of the domains of control, respectively by taking all resulting intervals and only those where strong control is proven.

Solver requirements. Both $SE_{S\&S}$ and $SE_{S\&SFB}$ incur the same solver requirements as SE_{SC}, with the addition of the *min* and *max* directives.

Comparison with Newsome et al.'s algorithm [53]. Our *S&S* algorithm bears some similarities with Newsome et al.'s feasible value set estimation algorithm. Both give a representation of what we call domains of control as a set of intervals, with a distinct splitting operation. Both also require *min* and *max* solver directives. However their algorithm splits intervals around random feasible values, which is fairly inefficient. Finally, they perform statistical sampling and compute a confidence interval on the density of each interval rather than proving strong control, leading to weaker guarantees.

On the other hand, our algorithm requires solving quantified SMT formulas and optimization queries, while theirs only issues standard quantifier-free SMT queries.

6.4 Conclusion

Table 1: Comparison of analysis methods for attacker control

Algorithm	TA	SE_{WC}	SE_{SC}	SE_{PMC}	Newsome	$SE_{S\&S}$
underlying problem(s)	-	QF SMT	Q SMT	PMC	QF SMT, min / max	Q SMT, min / max
$\neg WC$	✓	✓	✗	✓	✓	✓
WC	✗	✓	✓*	✓	✓	✓
SC	✗	✗	✓	✓	~	✓
QC	✗	✗	✗	✓	~	✓
DoC	✗	✗	✗	✗	~	✓
wQC	✗	✗	✗	✗	~	✓

QF SMT = Quantifier Free SMT, Q SMT = Quantified SMT, PMC = Projected Model Counting, min / max = directives for min and max values. ✓ means that an exact result can be given, ~ that only approximation can be given and ✗ that no result can be given.
* implicit when strong control is proven, but limited

As summed up in Table 1, we have shown that taint analysis cannot give much guarantees on attacker control. In comparison, all of our control properties can be proven or measured using algorithms based on symbolic execution. In particular, $SE_{S\&S}$ is able to compute domains of control, from which all other notions are derived.

7 Experimental Validation

We will now evaluate our approach and compare the characteristics of Shrink and Split with the other algorithms.

7.1 Implementation

We have implemented all the different techniques mentioned so far in a new generic binary-level dynamic analysis engine designed for flexibility and quick prototyping. We build on the Intel PIN binary instrumentation framework [5], the BINSEC symbolic execution engine [21, 24] and several external solvers. Our prototype is currently limited to x86_64 due to the use of PIN tools. Additionally:

- We perform taint analysis at the BINSEC IR level, allowing us to precisely track intra-instruction data-flows;
- SMT solving is performed via a portfolio approach with Z3 [22], Bitwuzla [54] in conjunction with various SAT solvers and Q3B [43];
- We support the exact PMC solvers d4 [47] and ganak [60] and the approximate solver approxmc [15, 62, 63] with $\mathcal{P}(result = truth \pm 20\%) = 0.95$ as a guarantee;
- We set a limit of k=100 splits for S&S[2];
- We reimplemented Newsome et al's algorithm based on the details provided in their paper [53].

The choice of precise yet architecture-specific binary-level analysis limits approximation in path constraints and thus ensures that the precision of control algorithms can be properly evaluated. However it is not an integral part of our method as any other way of deriving path constraints can be used, such as source level symbolic execution.

Automatically detecting and analyzing out-of-bounds memory accesses. We implemented an analysis based on taint which tracks pointers toward stack, heap and global objects. The taint information contains the bounds of the object and allows to check for violations during memory accesses. This allows to automatically identify and analyze the target variables for such vulnerabilities.

7.2 Benchmark

Our benchmark is split into a set of programs with well-understood vulnerabilities and another, more realistic one.

Ground-truth benchmark (B1). This benchmark is composed of the 14 real-world vulnerabilities from Table 2, plus 8 synthetic examples from Newsome et al. [53] and 17 new ones. The bugs in these programs were manually analyzed and are thus well understood. *The purpose of this benchmark will be to demonstrate the correctness of our approach.*

The real-world vulnerabilities in this benchmark were selected from the literature (*cve-2019-19307* and *cve-2019-14192* [28], heartbleed) and CVE databases. Regarding the latter, our requirements for open-source and reproducibility are surprisingly rarely fulfilled. The size of this benchmark is further limited by the need to establish ground truths manually and the lack of reuseable examples from the literature.

[2]The study in Appendix V shows a negligible impact of k on performance and precision above a certain threshold.

Table 2: Real-world vulnerabilities from the ground-truth benchmark B1 (see Appendices II and III for more details)

Program	Vulnerability name	type	Executed Instructions Symbolic	Total
libjpeg	cve-2023-37837	OOBR	56	261k
libsndfile	cve-2021-3246	OOBW	432	47k
mongoose	cve-2019-19307	IOF	104	67k
u-boot	cve-2019-14192	OOBW	38	4k
u-boot	cve-2019-14202	OOBW	38	4k
u-boot	cve-2022-30790	OOBW	161	3k
u-boot	cve-2022-30790-2*	OOBW	34	3k
u-boot	cve-2022-30552	OOBW	91	2k
openssl	heartbleed	OOBR	33	165 mil.
faad2	cve-2021-26567	CFH	14	12k
perl-dbi	cve-2020-14393	CFH	2491	60 mil.
sdop	cve-2024-41881	CFH	60	657k
xfpt	cve-2024-43700	CFH	271	179k
mjs	cve-2023-43338	CFH	99	51k

OOBR/W = Out-Of-Bounds Read / Write, IOF = Integer OverFlow, CFH = Control-Flow Hijacking
*side effect of cve-2022-30790

Table 3: Realistic benchmark B2 (see also Appendix IV)

Program	# of Vulnerabilities	Executed Instructions Symbolic	Total (million)
poppler	5	0 - 6671	5 - 15
openssl	9^1	91 - 5019	1 - 30
libtiff	5^2	274 - 1086	0.1 - 1
libxml	4	0 - 417	0.1 - 7
php	1	196	6
libpng	1	0	0.02
sqlite3	1	0	0.25

[1]including 6 variants not caught by Magma's ground truth oracles
[2]including 2 variants with different capabilities

Realistic benchmark (B2). This benchmark is composed of 26 out-of-bounds memory vulnerabilities from the Magma state-of-the-art fuzzing benchmark [33], in programs such as openssl, libtiff and libxml (see Table 3). They include buffer overflows, use-after-frees and various other invalid memory accesses. We picked these vulnerabilities based on whether they were triggered during the original evaluation of Magma.

These vulnerabilities were analyzed based solely on the available reproducing input and ASAN reports. Contrary to B1, no manual instrumentation was performed, hence the increase in realism toward in-the-field analysis of large, complex programs. Yet we lack a clear ground-truth for these vulnerabilities. *The purpose of this benchmark will thus be to demonstrate the practicality of our approach.*

7.3 Research Questions

We structure the experimental evaluation of our bug prioritization approach around the following research questions:

RQ1. How precise is Shrink and Split and how does it compare to other algorithms?

RQ2. How scalable is Shrink and Split and how does it compare to other algorithms?

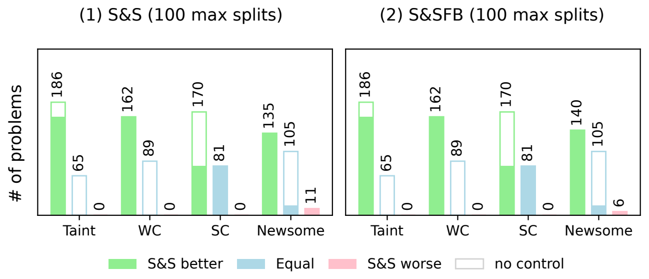
RQ3. How effective is our approach at prioritizing bugs compared to others?

RQ4. How does our approach fare in realistic end-to-end scenarios?

We chose a limit of 100 splits for S&S in our experiments. See Appendix V for a study of the split limit's impact.

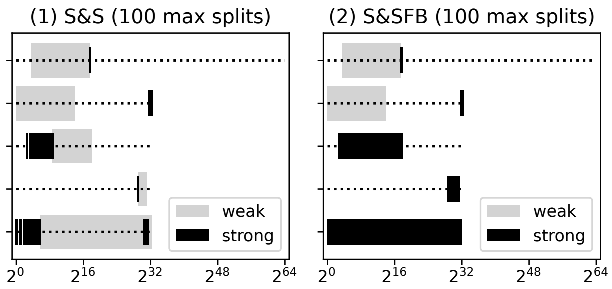
7.4 RQ1: Precision

We first compare the precision of S&S against other qualitative and quantitative methods.



Improvements in cases without control are due to 24 false positives for taint analysis and the fact that non-SC does not mean absence of control for SC.

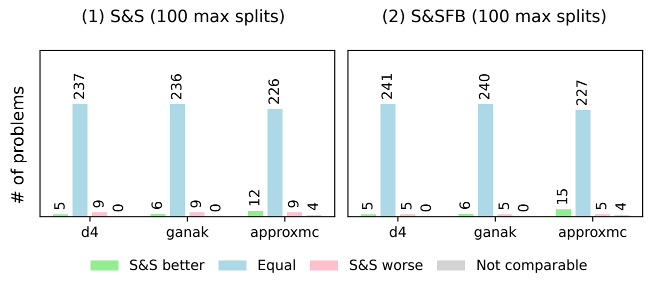
Figure 1: Precision of domains of control with S&S compared to other applicable algorithms



From top to bottom: cve-2021-3246.of_size, cve-2022-30790.wlen, cve-2022-30790_2.woff2, grub.canary, mul.j. See Appendix III for more examples. The dotted lines represent maximal value ranges based on variable sizes.

Figure 2: Examples of approximate results with S&S

Qualitative methods. The results are compared based on the inclusion of the returned domains of control, with smaller ones being considered better. Figure 1 shows that S&S is more precise than other applicable algorithm for measuring domains of control in most instances. In particular, it is strictly better than Newsome et al's algorithm on 135 (140 with S&SFB) cases out of 251, and worse in only 11 cases (6 with S&SFB). This is partly due to the loss of precision when the domains contain too many holes, which is mitigated by S&SFB (see Figure 2). On the other hand, S&S always improves upon WC when there is control, i.e., in 162 cases out of 251. It also beats taint analysis in these instances – note that tainting suffers from 24 false positives here. Finally, S&S can show absence of control while SC cannot, and also improves upon the latter in 81 cases with control.



Results from approxmc cannot be compared without an exact count.

Figure 3: Precision of quantitative control with S&S compared to model counters

Quantitative methods. We compare S&S to counting methods by reducing the domains to a single value count. It must be clear that S&S provides more information than quantitative methods by design, yet this comparison is restricted to counting only. Figure 3 details how S&S fares against d4 and ganak, as well as the approximate PMC solver approxmc. Overall S&S performs well in most instances, while S&SFB improves precision in 4 cases. Both algorithms also beat d4 and ganak on some problems due to them timing out. Finally, S&S and S&SFB beat approxmc in 12 and 15 cases.

Conclusion RQ1. While *S&S* may lose precision in some cases, it overall improves upon Newsome et al.'s algorithm and is even competitive against PMC solvers.

7.5 RQ2: Performance

Bug-prioritization methods can afford to be slower than bug-detection methods, as they only need to analyze buggy executions. Nevertheless, scalability always benefits usability.

Table 4 shows that S&S is competitive in terms of performance, with no time-outs and a total runtime of 11 minutes for both variants, beating its direct competitor Newsome et al.'s algorithm by a very large margin (2h09, 1 TO), with an average speedup of 73× for S&S and 77× for S&SFB[3]. This also shows that the additional fixed bits constraints does not meaningfully degrade performance. Counting approaches also suffer from time-outs.

[3]without the 5% top and bottom outliers

Table 4: Cumulative runtimes for each algorithm on B1+B2

Algorithm	Notion	Time-outs[1]	Runtime[2]
Taint	WC	0	< 0.1s
WC	WC	0	3s
SC	SC	0	46s
d4	QC	5	36m
ganak	QC	6	33m
approxmc	QC	1	15m
Newsome	DoC	1	2h09m
S&S[3]	DoC	0	11m
S&SFB[3]	DoC	0	11m

[1]5 minutes — [2]including time-outs — [3]100 maximum splits

Conclusion RQ2. Both *S&S* and *S&SFB* are significantly faster than the existing state-of-the-art and more robust than projected model counting algorithms.

7.6 RQ3: Bug priorization

So far we have evaluated the precision and scalability of Shrink and Split in relation to expectations and other algorithms. We will now show how our approach allows to precisely prioritize vulnerabilities via weighted quantitative control, with a case study on the 8 real-world buffer out-of-bounds vulnerabilities and our motivating example from our ground-truth benchmark B1. In addition, we also present a case study on 5 real-world control-flow hijack primitives (code pointer corruption). Its purpose is to illustrate the more simple data control problem.

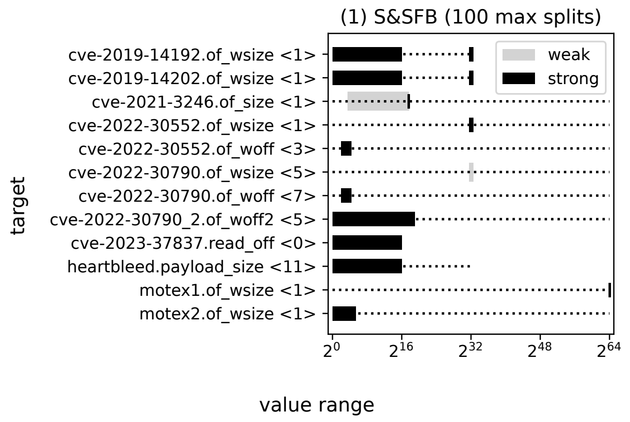


Figure 4: Domains of control for the OOB vulnerabilities

Distinguishing between different vulnerabilities. Table 5 details the CVSS, QC and wQC scores for each of the vulnerabilities in our study. These results show that weighted

Table 5: Using control to rate vulnerabilities

Out-Of-Bounds (memory range problem). *We rate control over memory ranges as the sum of the QC or wQC scores for offsets and sizes, multiplied by variable sizes. Non-out-of-bounds values are excluded and offsets are relative to the closest relevant bound for OOB write vulnerabilities. For wQC, we use the weight function* $\omega : x \mapsto \frac{1}{ln(2)x}$ *as in Section 5.4. Other similar functions yield equivalent results, see Appendix VII for more details.*

Control-Flow Hijacking (data problem). *We rate control over a corrupted code pointer. Since variable sizes are always 8 bytes, scores are normalized. When string parsing is involved, analysis takes into account individual byte constraints. For wQC, we give a null weight to invalid addresses, i.e., those corresponding to non-executable or unmapped memory. In particular, in 64-bit Linux, addresses with two non-null high bytes are invalid in user-space.*

Score categorization:
🙂 (low capabilities): CVSS < 5, OOB < 1, CFH < 0.01
😐 (medium capabilities): CVSS < 7, OOB < 10, CFH < 0.1
🙁 (high capabilities): CVSS ≥ 7, OOB ≥ 10, CFH ≥ 0.1

Vulnerability	CVSS		QC Score		wQC Score		Human[2]
OOB writes							
motex1	-		5.32	😐	0	🙂	🙂
motex2	-		5.32	😐	5.36	😐	😐
cve-2021-3246	8.8	🙁	17.6	🙁	14.14	🙁	🙁
cve-2019-14192	9.8	🙁	15.97	🙁	15.97	🙁	🙁
cve-2019-14202	9.8	🙁	15.97	🙁	15.97	🙁	🙁
cve-2022-30790	7.8	🙁	15.6	🙁	0.51	🙂	🙂
cve-2022-30552	5.5	😐	15.6	🙁	0.51	🙂	🙂
cve-2022-30790-2	-		15.94	🙁	2.37	😐	😐
OOB reads							
cve-2023-37837	6.5	😐	16	🙁	16	🙁	🙁
heartbleed	7.5	🙁	16	🙁	16	🙁	🙁
CFH							
cve-2021-26567	7.8	🙁	0.97	🙁	0	🙂	🙂
cve-2020-14393	7.1	🙁	0.99	🙁	1	🙁	🙁
cve-2024-41881	8.8	🙁	0.94	🙁	0	🙂	🙂
cve-2024-43700	7.8	🙁	0.78	🙁	0	🙂	🙂
cve-2023-43338	9.8	🙁	10^{-5}	🙂	1	🙁	🙁
Total Correct	6/12		7/15		15/15		

[1]7 for CVSS — [2]manual analysis performed by us

quantitative control, and by extension domain of control analysis, is able to clearly differentiate between vulnerabilities with different capabilities, such as *motex1* and *motex2* (resp. vulnerabilities a and b from Section 2) or *cve-2022-30552* and *cve-2019-14192*, with the former only allowing for very large write sizes (see Figure 4) and thus being less dangerous.

Identical capabilities. Additionally, domains of control allow to recognize vulnerabilities sharing the same capabilities, i.e., those with the same target variables and domains of control, such as *cve-2019-14192* and *cve-2019-14202* or *cve-2022-30790* and *cve-2022-30552* (see Figure 4).

Correcting human analysis. In the case of *cve-2022-30790*, we found previous human analysis to be incorrect. It was discovered together with *cve-2022-30552* [6] and was thought to grant an arbitrary write primitive by overwriting metadata in a linked list. In contrast, the other can only be used for DOS attacks due to a very large write size caused by an integer underflow. However, our results contradict this interpretation as both vulnerabilities are found to have the same capabilities, matching the analysis of *cve-2022-30552*. Inspecting the code reveals that some security checks prevent most out-of-bounds writes, which must have been missed by previous evaluators (see Appendix VIII). Using our tool could have helped them realize their mistake and thus improve human analysis.

Comparing control-flow hijacking primitives. The main point of interest regarding these vulnerabilities is the fact that QC is particularly misleading. This is due to the fact that the two upper bytes of the corrupted code pointer must be set to zero in order to prevent a crash. As a result, vulnerabilities with high QC such as *cve-2021-26567* and *cve-2024-41881* are not exploitable since the data overwriting the code pointer comes from a string, hence bytes cannot be null.

In contrast, *cve-2023-43338* has low QC but is very controllable since any value between 0 and $2^{48} - 1$ can be obtained (see Appendix III).

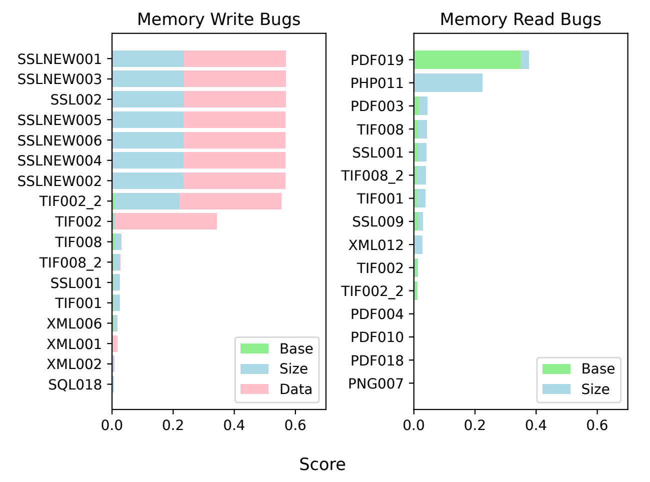
Comparison with other methods. CVSS and quantitative control scores are not so useful as differences are too minor to be meaningful. This is to be expected for the former as it is only a loose indicator assuming worst case. On the other hand, quantitative control cannot account for differing value threat levels. Weak and strong control alone are also not very useful as in most cases control is weak but not strong. Overall, weighted QC scores are the only ones to match our expectations, with 12/12 matches for the vulnerabilities with CVSS scores, against 6/12 for the latter and QC.

Conclusion RQ3. Our approach allows to prioritize vulnerabilities more precisely than the existing state-of-the-art in our case study. It would also have improved human analysis for *cve-2022-30790*.

7.7 RQ4: Realistic end-to-end scenarios

To demonstrate the practicality of our approach, we automatically analyzed and rated the Magma vulnerabilities from benchmark B2, starting from a single input only and with practically no human effort. Figure 5 displays the resulting scores, which clearly differentiate vulnerability capabilities. Overall, our wQC scores match expectations and allows us to derive interesting knowledge of the vulnerabilities (see Appendix IX for a more in-depth discussion).

OOB writes. We observe a correlation with the 11 available CVSS scores, with 9 out of 11 close matches with our wQC score and a clear distinction between highly and less controllable vulnerabilities. Regarding the two mismatches, *SSL001* and *TIF001*, we show that despite their high CVSS scores,



Scores are computed for base addresses and sizes using weighted quantitative control biased toward local OOBs. Our weight function is $x \mapsto \frac{1}{ln(2)d(x)}$ with d the distance between x and the nearest buffer bound for addresses or minimum out-of-bounds size for sizes. Data scores are given as an average of quantitative control over the first eight written bytes.

Figure 5: OOB capability scores for the Magma bugs

their capabilities are actually very limited, since they consists in repeated non-contiguous single byte writes, with no control over any parameters.

We also observe a similarity between the *SSLNEW* vulnerabilities' capabilities and those of *SSL002*. The fact that they are fixed by the same patch suggests that they are indeed most likely derived from the same bug.

OOB reads. We observe the same trend as OOB writes, although some CVSS scores are influenced by the occurrence of OOB writes in the same execution, complicating the interpretation. Additionally, our analysis suggests that *PDF010*, which does not have an associated CVE, has the same capabilities as three other vulnerabilities with CVEs.

Conclusion RQ4. Our approach can be fully automated and still clearly differentiate highly controllable vulnerabilities from others in realistic fuzzing targets.

8 Discussion

Handling Other Types of Vulnerabilities. In our experimental validation, we focused on memory out-of-bounds vulnerabilities. However our method can be applied to other types of vulnerabilities as well. In particular, vulnerabilities resulting in control over well-defined data value (data control problem model) can be handled by computing weighted quantitative control over said value with an appropriate weight function. For example, regarding pointer corruption and assuming a 64-bit architecture, the weight function should attribute a weight of zero to any value greater or equal to 2^{48}, since those addresses are invalid.

On the other hand, our tool could also be used to measure information leakages, since they are typically evaluated using data-flow analysis. For example, in our heartbleed experiment, we analyzed 8 bytes of unencrypted data leaked from a previous server interaction. Yet indirect leakages (e.g., side-channels) would be difficult to handle.

Finally, our method may not be applicable to some kinds of vulnerabilities (e.g., DoS caused by infinite loops) and may be difficult to leverage for others (e.g., heap layout manipulation, although we discussed some possibilities in Section 4).

Multi-path analysis and implementation. While single path analysis simplifies symbolic execution and greatly improves scalability by avoiding path explosion, it can result in a loss of completeness. A possible mitigation is to explore additional paths and merge those reaching the same vulnerability, yet it should be done with care in order to avoid any significant performance cost. Another solution would be to explore multiple paths separately, then merge the domains of control.

Note that our framework, algorithms and general approach does not rely on any particular constraint computation technique, thus other implementations may choose to sacrifice performance for more completeness with multi-path analysis. We also investigated how standard symbolic execution optimizations impact our approach. As expected, constraint relaxation [55] (over-approximation) yields wider domains (see Appendix XI) while partial input concretization [20, 30] (under-approximation) induces tighter ones (see Appendix XII). Both may reduce computation time but also impact bug evaluation. A systematic evaluation is left as future work.

Interpreting results. While weighted quantitative control can help to automatize vulnerability evaluation, expert insight is still required in order to define a suitable weight function. This can also arguably be an advantage, as it allows to tailor the method toward the specific needs and constraints of different projects, hardware or environments.

On a different note, weak intervals in domains computed by Shrink and Split can be difficult to interpret, as they may have very different densities (see Appendix X). Fortunately, our threat classification from Table 5 is unaffected, but this may not always be the case.

Limits of automation. Our tool automation is enough to handle many situations, as illustrated in Section 7.7. However, it cannot track some implicit properties, such as constraints on the length of strings. Repeated memory accesses inside loops can also be tricky to properly analyze, as some parameters may overall be tied to the looping condition. These are universal problems in binary-level analysis.

Limits of control for exploitability evaluation. Exploitability is a fundamentally hard to capture concept. As such, one can only aspire to identify and evaluate some of its aspects, such as control. This also means that any formal exploitability assessment technique is limited by the scope of its target property. In our case, control measurements may be too narrow or approximate due to the technical limitations of our implementation, but factors other than control may also be at play, such as a lack of robustness [29].

Binary vs. source code analysis for prioritization. Given the variety of existing platforms, environments and building tools, source-level analysis may appear a good choice for generic bug evaluation. Yet, bugs of the types we consider here are often found at the binary level (e.g., ASAN detections during fuzzing) and their exploitation often depends on low-level concepts such as memory layouts. They are thus difficult to precisely characterize at source level.

9 Related Works

Bug Impact Evaluation. Manual bug prioritization efforts such as CVSS scores tend to always assume worst case scenarios, regardless of the actual capabilities of vulnerabilities. In addition, human error is an ever-present risk as illustrated by the case of *cve-2022-30790* (Section 7.6).

On the other hand, automated bug prioritization practices usually consist in attributing different priority levels to different types of vulnerabilities. In the case of fuzzing, sanitizers such as ASAN [59] or KASAN [4] provide information on bug impacts. While it works well for coarse-grained prioritization, it does not allow to distinguish between vulnerabilities of the same type.

Syzscope [71] refines this approach by allowing the execution to continue after a first issue is detected, potentially leading to more serious ones. While this allows to identify dangerous bugs which would otherwise be considered benign, it still does not allow to distinguish between vulnerabilities of the same type.

Evocatio [42] uses targeted fuzzing to discover additional bug capabilities, such as different sizes or offsets of out-of-bounds writes, but does not score nor rank them.

KOOBE [16] characterizes out-of-bound reads and writes using symbolic execution in order to identify the most promising ones to automatically build exploits. It ranks capabilities only when their associated constraints are either identical, or one of them is a constant value and solution of the other (partial order). While useful in an automatic exploit generation setting, this approach is too simple for full-fledged bug prioritization: no two vulnerabilities from Table 5 could be compared with it. Despite its limitations, KOOBE is to our knowledge the closest existing work to offering a generic fine-grained vulnerability capability metric.

ML-based methods. Existing works attempt to predict the exploitability of vulnerabilities using deep learning techniques [19, 41, 48, 64]. While these approaches scale well once trained, the lack of transparency in the results as well as the over-reliance on human analysis (bug reports, CVSS scores, etc.) currently hinder their usability.

Automatic Exploit Generation (AEG). One way to prove the exploitability of a bug is to build an exploit around it. However, building exploits manually is difficult, hence the interest in automatic exploit generators (AEG). While first attempts focused on shellcode injection exploits [8, 14, 34], more recent works focus on heap exploitation [25, 35, 36, 56, 66, 67, 70], kernel exploitation [16, 17, 49, 68, 69] or gadget chain synthesis [38–40, 58].

While a priori close in goals, AEG is not well suited for bug priorization. Indeed, AEG tools tend to be highly specialized, so that while building an exploit is the strongest possible proof of exploitability, failure is less conclusive, as the vulnerability could require a type of exploit not covered by the AEG engine at hand. AEG may also involve fuzzing for similar-yet-different vulnerabilities [16, 66, 69].

Robust Reachability. Another aspect of the exploitability of bugs besides their security impact is how reliably they can be triggered. Girol et al. [27–29] proposed robust reachability to formally capture this notion that complements but is orthogonal to bug impact analysis for prioritization.

Quantitative Information Flow. Measuring channel capacity and thus quantitative information flow has mainly been developed for the purpose of quantifying leakage of secret information [11,37,51]. These works consider notions similar to quantitative control, while we argue that qualitative domains of control are a much better tool for bug prioritization.

Newsome et al. [53] use quantitative information flow methods to measure quantitative control. Their algorithm internals share similarities with S&S, even though their goal is purely quantitative and they do not identify the key notion of domains of control. Additionally, their algorithm has not been designed nor used for bug prioritization, and our experiments show that S&S performs much better for our needs.

10 Conclusion

We focused on the problem of precise and efficient bug prioritization, with the expressed goal of distinguishing more or less security-critical bugs. Our work on the evaluation of attacker control over vulnerability parameters to distinguish between vulnerabilities constitutes a step in this direction, yielding a theoretical framework and efficient analysis methods. In summary, we argue that attacker control analysis should focus on domains of control, as it allows to account for finer-grained threat models. Our "Shrink and Split" algorithm yields said domains of control in a scalable and flexible manner with strong formal guarantees, lending itself to practical use as shown in our experiments on real-world programs. Future efforts could focus on applying our approach to a wider array of vulnerabilities.

Ethical Considerations

All vulnerabilities discussed in this work are known and/or patched in the corresponding software's current version. On the other hand, if efficient and precise bug prioritization would be very beneficial to bug-fixing efforts by developers, it could also be used by malicious actors to find more promising bugs to exploit. Nevertheless, we argue that our approach constitutes an improvement in that regard over automated exploit generation, as it does not directly enable low-skill attackers to wield ready-made exploits.

Acknowledgements

This work was partially supported by the "France 2030" government investment plan managed by the French National Research Agency, under the references ANR-22-PECY-0009 and ANR-22-PECY-0005.

Availability

All our research artifacts are openly available at https://doi.org/10.5281/zenodo.14699098. This includes the source code of our tool Colorstreams, nix-based packaging for reproducible compilation and easy management of dependencies such as BINSEC, a docker image for easy deployment, both of our evaluation benchmarks with scripts automating the reproduction of experiments and generating figures, user tutorials and API documentation for developers.

References

[1] https://syzkaller.appspot.com/upstream. Online, accessed December 18th 2024.

[2] https://github.com/google/syzkaller. Online, accessed June 27th 2023.

[3] https://github.com/google/oss-fuzz?tab=readme-ov-file. Online, accessed December 18th 2024.

[4] https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html. Online, accessed June 27th 2023.

[5] https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. Online, accessed June 28th 2023.

[6] https://research.nccgroup.com/2022/06/03/technical-advisory-multiple-vulnerabilities-in-u-boot-cve-2022-30790-cve-2022-30552/. Online, accessed October 4th 2023.

[7] https://github.com/u-boot/u-boot/blob/v2022.01/net/net.c. Online, accessed December 13th 2024.

[8] Thanassis Avgerinos, Sang Cha, Brent Hao, and David Brumley. Aeg: Automatic exploit generation. NDSS 2011.

[9] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time Markov chains. CAV 1996.

[10] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. Projected model counting. SAT 2015.

[11] Fabrizio Biondi, Michael Enescu, Annelie Heuser, Axel Legay, Kuldeep Meel, and Jean Quilbeuf. Scalable approximation of quantitative information flow in programs. VMCAI 2018.

[12] Nikolaj S. Bjørner and Anh-Dung Phan. νz - maximal satisfaction with z3. SCSS 2014.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. USENIX OSDI 2008.

[14] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. S&P 2012.

[15] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. IJCAI 2016.

[16] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. USENIX Security 2020.

[17] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. CCS 2019.

[18] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. QAPL 2001.

[19] Siddhartha Shankar Das, Edoardo Serra, Mahantesh Halappanavar, Alex Pothen, and Ehab Al-Shaer. V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities. DSAA 2021.

[20] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. ISSTA 2016.

[21] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. SANER 2016.

[22] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. TACAS 2008.

[23] Dorothy Denning. Cryptography and data security. *SERBIULA (sistema Librum 2.0)*, 1982.

[24] Adel Djoudi and Sébastien Bardin. BINSEC: binary code analysis with low-level regions. TACAS 2015.

[25] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. USENIX Security 2018.

[26] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. ISSTA 2012.

[27] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Introducing robust reachability. *Formal Methodes in System Design (FMSD)*, 2022.

[28] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not all bugs are created equal, but robust reachability can tell the difference. CAV 2021.

[29] Guillaume Girol, Guilhem Lacombe, and Sébastien Bardin. Quantitative robustness for vulnerability assessment. PLDI 2024.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. PLDI 2005.

[31] J. A. Goguen and J. Meseguer. Security policies and security models. S&P 1982.

[32] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5), 1994.

[33] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), 2020.

[34] Sean Heelan and Daniel Kroening. Msc computer science dissertation automatic generation of control flow hijacking exploits for software vulnerabilities. 2009.

[35] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. CCS 2019.

[36] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. USENIX Security 2018.

[37] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. ACSAC 2010.

[38] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. USENIX Security 2015.

[39] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. S&P 2016.

[40] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. CCS 2018.

[41] Yuning Jiang and Yacine Atif. An approach to discover and assess vulnerability severity automatically in cyber-physical systems. SIN 2020.

[42] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring bug capabilities from a single poc. CCS 2022.

[43] Martin Jonás and Jan Strejcek. Solving quantified bit-vector formulas using binary decision diagrams. SAT 2016.

[44] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Xiaodong Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. NDSS 2011.

[45] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. ICSE 2009.

[46] Vladimir Klebanov. Precise quantitative information flow analysis— a symbolic approach. *Theoretical Computer Science*, 538, 2014.

[47] Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. IJCAI 2017.

[48] Triet H. M. Le, Huaming Chen, and M. Ali Babar. A survey on data-driven software vulnerability assessment and prioritization. *ACM Comput. Surv.*, 55(5), 2022.

[49] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. NDSS 2017.

[50] Sanoop Mallissery and Yu-Sung Wu. Demystify the fuzzing methods: A comprehensive survey. *ACM Comput. Surv.*, 56(3), 2023.

[51] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. *SIGPLAN Not.*, 43(6), 2008.

[52] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. USENIX Security 2015.

[53] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. PLAS 2009.

[54] Aina Niemetz and Mathias Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020.

[55] David A. Ramos and Dawson Engler. Under-Constrained symbolic execution: Correctness checking for real code. USENIX Security 2015.

[56] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. Modular synthesis of heap exploits. PLAS 2017.

[57] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). S&P 2010.

[58] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. USENIX Security 2011.

[59] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. USENIX ATC 2012.

[60] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. IJCAI 2019.

[61] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. S&P 2016.

[62] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling.

[63] Mate Soos and Kuldeep S. Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. AAAI 2019.

[64] Octavian Suciu, Connor Nelson, Zhuoer Lyu, Tiffany Bao, and Tudor Dumitras. Expected exploitability: Predicting the development of functional vulnerability exploits. USENIX Security 2022.

[65] Chenghua Tang, Xiaolong Guan, Mengmeng Yang, and Baohua Qiang. Taintse: Dynamic taint analysis combined with symbolic execution and constraint association. ICSESS 2023.

[66] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. CCS 2018.

[67] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards automated heap feng shui. USENIX Security 2021.

[68] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. USENIX Security 2019.

[69] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. USENIX Security 2018.

[70] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. USENIX Security 2020.

[71] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel. USENIX Security 2022.

Appendix I. Our taint propagation model

expression	target	taint value
basic rules (lattice: $\bot$ — $\top$)		
$v = op\ e$	v	$t(e)$
$v = e_1\ op\ e_2$	v	$t(e_1) \sqcup t(e_2)$
$store(addr, e)$	$mem[addr]$	$t(e)$
$v = load(addr)$	v	$t(mem[addr])$
$source(v)$	v	$\top$
(*option*) propagating control-flow dependencies*		
$if(cond)\ do\ v = e$	v	$t(cond) \sqcup t(e)$
(*option*) over-approx memory operations when $t(addr) = \top$		
$store(addr, e)$	mem	$t(e)$
$v = load(addr)$	v	$\sqcup_{addr} t(mem[addr])$
(*option*) local taint suppression rules (non exhaustive)		
$v = e \times 0$	v	$\bot$
$v = e - e$	v	$\bot$
$if(v = const)$	v	$\bot$

*Here we present a very straightforward version of control-flow dependency propagation – more precise and practical approaches such as DTA++ only propagate taint in a few selected relevant cases [44].

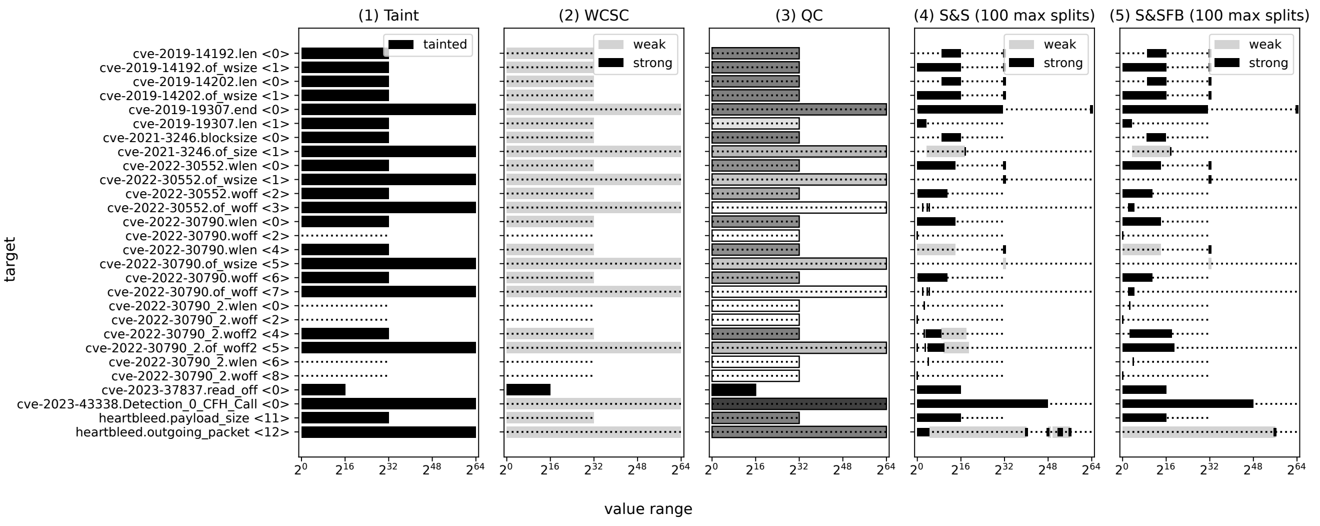
Variable and expressions are either untainted ($\bot$) or tainted ($\top$). $\sqcup$ is the merge operator over taint values ($\top \sqcup \bot = \top$). Taint is introduced at *sources*, such as program inputs. Memory is represented as an array named *mem*. Optionally, one may propagate taint from jump conditions to subsequent assignments or from memory addresses to written or read data in memory operations. Taint can also be suppressed when a single instruction reduces a tainted expression to a single value. The lack of constraint computation and the per-instruction granularity does not allow to propagate taint based on multi-instruction behaviours.

Appendix II. Detailed ground-truth benchmark B1

Program	Vulnerability	Targets	Control	Executed Instructions	
	Type(s)		Problem	Symbolic	Total
Toy examples from Newsome et al. [53]					
copy	-	V	D	2	22
ccopy	-	V	D	6	27
mcopy	-	V	D	3	23
mul	-	V	D	3	23
div	-	V	D	3	23
impflow	-	V	D	3	40
mixdup	-	V	D	13	33
popcnt	-	V	D	39	59
Other toy examples					
sum	-	V	D	4	32
sub	-	V	D	3	24
motex1 (vuln. a)	IUF, OOBW	S	MRW	18	78
motex2 (vuln. b)	UBI, OOBW	S	MRW	15	1848
koobe (Chen et al. [16])	TC, PC	F	D	12	4075
uafubi	UAF, UBI, PC	F	D	11	2851
uafubi2	UAF, UBI, PC	F	D	2	2843
spray	UBI	D	F	8	2224
spray2	UBI	D	F	16	2248
spray3	UBI	D	F	20	89
can	OOBW, PC	V, F	D	2	2709
can2	OOBW, PC	V, F	D	3	2830
grub ($\sim$ cve-2015-8370)	OOBW	V	D	84	3868
cfi	-	F	D	12	2249
cfi2	-	F	D	5	2218
minesweeper1	OOBW, PC	R	D	278	63903
minesweeper2	OOBW	I	MRW	254	103315
Real vulerabilities					
cve-2023-37837 (libjpeg)	OOBR	I	MRR	56	261512
cve-2021-3246 (libsndfile)	OOBW	S	MRW	432	46872
cve-2019-19307 (mongoose)	IOF	V	D	104	67345
cve-2019-14192 (u-boot)	IUF, OOBW	S	MRW	38	4296
cve-2019-14202 (u-boot)	OOBW	S	MRW	38	4296
cve-2022-30790 (u-boot)	OOBW	I, S	MRW	161	3465
cve-2022-30790-2 (u-boot)	OOBW	I	MRW	34	3173
cve-2022-30552 (u-boot)	IUF, OOBW	I, S	MRW	91	1532
heartbleed (openssl)	OOBR	S	MRR	33	165388032
cve-2021-26567 (faad2)	OOBW, PC	R	D	14	12799
cve-2020-14393 (perl-dbi)	OOBW, PC	R	D	2491	59435770
cve-2024-41881 (sdop)	OOBW, PC	R	D	60	657357
cve-2024-43700 (xfpt)	OOBW, PC	R	D	271	179369
cve-2023-43338 (mjs)	PC	F	D	99	51401

vulnerability types: UAF = use-after-free, UBI = use-before-initialization, OOBR / OOBW = out-of-bounds read / write, TC = type confusion, PC = pointer corruption, IUF / IOF = integer underflow / overflow — **targets:** R = return address, F = function pointer, V = stack variable, I = index / offset, S = size — **control problems:** D = data, MRR / MRW = memory range read / write

Appendix III. Domains of control for the ground-truth benchmark B1 CVEs (single-bytes excluded)



Appendix IV. Detailed realistic benchmark B2

Bug	Vulnerability	Mapping	Executed Instructions	
	Type(s)		Symbolic	Total
PDF003	OOBR	stack	0	5099250
PDF004	OOBR	other	6671	7371051
PDF010	OOBR	other	0	14997856
PDF018	OOBR	other	0	13291270
PDF019	OOBR	heap	1463	7069344
PHP011	OOBR	heap	196	6059203
PNG007	OOBR	other	0	20971
SQL018	OOBW	heap	0	251339
SSL001	OOBR, OOBW	heap	5019	1448773
SSL002	UAFW	heap	295	10384861
SSL009	OOBR	heap	3580	6351194
SSLNEW001	UAFW	heap	91	10408709
SSLNEW002	UAFW	heap	104	12883298
SSLNEW003	UAFW	heap	284	10374848
SSLNEW004	UAFW	heap	105	29008645
SSLNEW005	UAFW	heap	105	29024109
SSLNEW006	UAFW	heap	105	29133273
TIF001	OOBR, OOBW	heap	1086	218407
TIF002	OOBR, OOBW	heap	624	981495
TIF002_2	OOBR, OOBW	heap	274	960672
TIF008	OOBR, OOBW	heap	508	169138
TIF008_2	OOBR, OOBW	heap	734	238135
XML001	OOBW	stack	417	612662
XML002	OOBW	heap	0	147239
XML006	OOBW	stack	0	639552
XML012	UAFR	heap	0	7019269

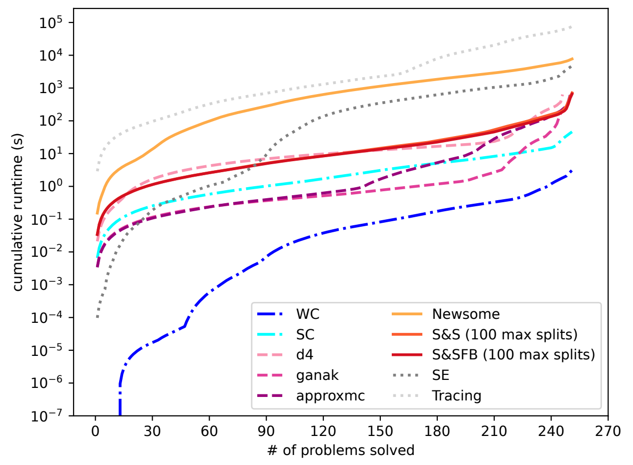
vulnerability types: UAFR / UAFW = use-after-free read / write, OOBR / OOBW = out-of-bounds read / write

Appendix V. Results of Shrink and Split with different split limits on B1+B2

Algorithm	Splits	Notion	Exact	Approx. ($< \times 2$)	Runtime
S&S	10	DoC	235	11 (6)	10m56s
S&S	50	DoC	237	9 (4)	11m12s
S&S	100	DoC	237	9 (4)	11m52s
S&S	500	DoC	237	9 (4)	15m20s
S&S	1000	DoC	237	9 (4)	19m33s
S&SFB	10	DoC	238	8 (7)	10m26s
S&SFB	50	DoC	240	6 (5)	10m45s
S&SFB	100	DoC	241	5 (4)	10m55s
S&SFB	500	DoC	241	5 (4)	13m05s
S&SFB	1000	DoC	241	5 (4)	15m29s

Increasing the split limit for Shrink and Split has a limited impact on precision on our benchmark. Since holes are likely to occur at regular intervals in cases where they are numerous, other solutions such as taking fixed bits into account improve precision more effectively. Higher split limits only affect runtimes marginally.

Appendix VI. Cactus plot of all tested algorithms on B1+B2



Tracing and symbolic execution are common to all other algorithms. The sum of both indicates the time needed to derive path constraints.

Appendix VII. Testing multiple weight functions for scoring the OOB vulnerabilities from Table 5

Vulnerability	$\omega : x \mapsto \frac{1}{ln(2)x}$		$\omega' : x \mapsto \frac{1}{x^2}$		$\omega'' : x \mapsto \frac{1}{\sqrt{x}}$	
			OOB writes			
motex1	0	🙂	0	🙂	0	🙂
motex2	5.36	😐	15.23	🙁	7.9e-8	😐
cve-2021-3246	14.14	🙁	5.33	😐	6.5e-6	😐
cve-2019-14192	15.97	🙁	16	🙁	0.12	🙁
cve-2019-14202	15.97	🙁	16	🙁	0.12	🙁
cve-2022-30790	0.51	🙂	1.75	🙂	8.4e-9	🙂
cve-2022-30552	0.51	🙂	1.75	🙂	8.4e-9	🙂
cve-2022-30790-2	2.37	😐	8	😐	1.3e-6	😐
			OOB reads			
cve-2023-37837	16	🙁	16	🙁	16	🙁
heartbleed	16	🙁	32	🙁	0.12	🙁

Compared to ω, ω' favors smaller values more while ω'' is less biased toward them. This results in some differences, especially regarding *motex2* and *cve-2021-3246*. However the general tendency remains similar overall.

Appendix VIII. Manual Evaluation of *cve-2022-30790* and *cve-2022-30552*

Both vulnerabilities occur in uboot v2022.01, in the *__net_defragment* function from *net/net.c*, line 900 [7]. This function takes packets containing data fragments and puts them back together into a static buffer. A linked list structure is used to keep track of holes in the data and check if the incoming fragments fit into them.

```
 1  static struct ip_udp_hdr *__net_defragment
 2      (struct ip_udp_hdr *ip, int *lenp)
 3  {
 4      /*static buffer where the data is assembled*/
 5      static uchar pkt_buff[IP_PKTSIZE];
 6      ...
 7      /*hole linked list metadata struct*/
 8      struct hole *payload, *thisfrag, *h, *newh;
 9      ...
10      uchar *indata = (uchar *)ip;
11      int offset8, start, len, done = 0;
12      /*data fragment offset from packet header*/
13      u16 ip_off = ntohs(ip->ip_off);
14
15      /*start of data in pkt_buff*/
16      payload = (struct hole *)(pkt_buff
17          + IP_HDR_SIZE);
18      offset8 =  (ip_off & IP_OFFS);
19      /*start of incoming fragment in pkt_buff*/
20      thisfrag = payload + offset8;
21      start = offset8 * 8;
22      /*data fragment length computation*/
23      /*can go negative (cve-2022-30552)*/
24      len = ntohs(ip->ip_len) - IP_HDR_SIZE;
25
26      /*Here the program checks if the data fragment
27      would overflow pkt_buff. However a negative
28      len can pass if start >= IP_HDR_SIZE.*/
29      if (start + len > IP_MAXUDP)
30          return NULL;
31
32      ...
33      /*Here the function checks if the packet is
34      coherent with already received fragments, i.e.,
35      if it fits into a hole in pkt_buff. These holes
36      are tracked via metadata stored into them akin
37      to a linked list. If there is a conflict, the
38      function aborts.*/
39      /*The function also updates the metadata,
40      which can be corrupted (cve-2022-30790).*/
41      /*thisfrag and len are NOT updated.*/
42      ...
43
44      /*thisfrag and len are independent from the
45      hole metadata, which is only used to auth-
46      orize the write. Therefore an overflow can
47      only happen if len is negative and underflows
48      due to an implicit cast to size_t
49      (cve-2022-30552).*/
50      memcpy((uchar *)thisfrag, indata + IP_HDR_SIZE,
51          len);
52      ...
53  }
```

Listing 4. Vulnerable __net_defragment function from uboot v2022.01

Listing 4 gives a summary of the manual analysis of *cve-2022-30552* and *cve-2022-30790* and explains why *cve-2022-30790* cannot lead to arbitrary writes outside of the target buffer on line 50 as was previously thought [6]. In particular, the linked list metadata corruption cannot lead to bypassing the check on line 29. Our tool shows the same capabilities for both vulnerabilities due to *cve-2022-30552* being triggerable within *cve-2022-30790*'s execution path.

On the other hand, *cve-2022-30790* enables some limited out-of-bounds writes during the linked list update, which were not discussed in the original human analysis. We analyzed those capabilities as *cve-2022-30790-2*.

Appendix IX. Magma Vulnerability Scores Discussion

Our results for the Magma vulnerabilities from RQ4 (Section 7.7) illustrate the benefits of our approach as well as areas of possible improvements.

Table 6: Detailed scores for the magma vulnerabilities (B2)

Bug	CVE	CVSS	wQC Score			
			Base	Size	Data	Overall
			OOB writes			
SSLNEW001	-	-	0.00477	0.703	1	0.569
SSLNEW003	-	-	0.00477	0.703	1	0.569
SSL002	CVE-2016-6309	9.8	0.00477	0.703	1	0.569
SSLNEW005	-	-	0.00473	0.697	1	0.567
SSLNEW006	-	-	0.00473	0.697	1	0.567
SSLNEW004	-	-	0.00473	0.697	1	0.567
SSLNEW002	-	-	0.00469	0.696	1	0.567
TIF002_2	CVE-2016-5314	8.8	0.022	0.643	1	0.555
TIF002	CVE-2016-5314	8.8	0.0223	0.00512	1	0.342
TIF008	CVE-2015-8784	6.5	0.0285	0.0587	0.00391	0.0304
TIF008_2	CVE-2015-8784	6.5	0.02	0.0587	0.00391	0.0275
SSL001	CVE-2016-2108	9.8	0.0208	0.0554	0.00391	0.0267
TIF001	CVE-2016-9535	9.8	0.0192	0.0554	0.00391	0.0262
XML006	CVE-2017-9048	7.5	0.0173	0.0342	0.00391	0.0185
XML001	CVE-2017-9047	7.5	3.27e-05	6.76e-05	0.0508	0.017
XML002	CVE-2017-0663	7.8	0.000912	0.0178	0.00391	0.00755
SQL018	CVE-2015-3414	7.5	0.00398	0.0102	0.00391	0.00602
			OOB reads			
PDF019	CVE-2017-9776	7.8	0.699	0.0535	0	0.376
PHP011	CVE-2018-14883	7.5	0.000181	0.447	0	0.224
PDF003	CVE-2017-9865	5.5	0.0303	0.0585	0	0.0444
TIF008	CVE-2015-8784	6.5*	0.0285	0.0587	0	0.0436
SSL001	CVE-2016-2108	9.8*	0.027	0.0554	0	0.0412
TIF008_2	CVE-2015-8784	6.5*	0.02	0.0587	0	0.0393
TIF001	CVE-2016-9535	9.8*	0.0192	0.0554	0	0.0373
SSL009	CVE-2017-3735	5.3	0.0287	0.0313	0	0.03
XML012	CVE-2016-1836	5.5	0.000277	0.0537	0	0.027
TIF002	CVE-2016-5314	8.8*	0.0223	0.00512	0	0.0137
TIF002_2	CVE-2016-5314	8.8*	0.022	0.00104	0	0.0115
PDF004	CVE-2019-10873	6.5	0	0	0	0
PDF010	-	-	0	0	0	0
PDF018	CVE-2018-10768	6.5	0	0	0	0
PNG007	CVE-2013-6954	5	0	0	0	0

*likely based on OOB writes
grey lines represent score cutoff points

Contrast between different vulnerability capabilities. At a glance, our results from Figure 5 and Table 6 show a clear difference in OOB capabilities between the vulnerabilities. While it is not feasible to analyze each vulnerability in depth, we can still check that the scores are appropriate by manually reviewing and interpreting the domains of control and other readily available information.

On the side of OOB writes, we can see a clear discrepancy between highly controllable and uncontrollable vulnerabilities. In particular, *SSL002* and the *SSLNEW* vulnerabilities grant attackers the ability to write arbitrary data over a controlled amount of bytes, starting at a static address. This alone makes these vulnerabilities quite dangerous and grants them a high score, but furthermore, we know that they are heap use-after-frees, as shown in Appendix IV. Hence the base address of the write could be indirectly influenced via heap layout manipulation.

In comparison, *TIF002* only grants control over the written data, which could still be useful to an attacker, hence its middling score. However, other vulnerabilities such as *TIF001* cannot be controlled much, if at all, and have a low score as a result. They could only be exploited if, by luck, they align perfectly with exploit requirements. *TIF001* in particular is an *off-by-one* OOB write which does not allow control over the written data.

Regarding OOB reads, *PDF019* and *PHP011* offer the widest capabilities, although this is arguably less important for these vulnerabilities to be exploitable, since useful information can be extracted from a large read and overwriting critical data in a detrimental way is not a concern. Still, our results show that *PDF004*, *PDF010*, *PDF018* and *PNG007* cannot be valid writes and thus can only be used to cause crashes. Their score is, appropriately, zero.

Correlation with CVSS scores. While CVSS scores are worst-case and qualitative at best, it is interesting to note that we still observe a light correlation between them and our scores, especially for OOB writes. For those, we only find two outliers, *SSL001* and *TIF001*, with high CVSS scores but low wQC scores. Both analyzed writes are *off-by-one* without controlled data, and thus would be difficult to exploit. However, note that in both cases we only included results for the first write out of several, which all exhibit roughly the same characteristics and are not contiguous as could be expected from a loop. Combining all information could results in higher capabilities, however the lack of control on the written data is a hard limit on exploitability.

For OOB reads, some vulnerability also cause OOB writes, which explains their higher scores. It is also interesting to note that *PDF010* grants the same capabilities as *PDF004*, *PDF018* and *PNG007*, the ability to cause crashes with an invalid read, yet does not have an associated CVE.

Analysis of new vulnerabilities. The *SSLNEW* vulnerabilities are not technically part of the magma benchmark. They were triggered during the original Magma evaluation but not caught by any of Magma's bug oracles. From our results, we can see that they have very similar capabilities to *SSL002*, as well as the same type, namely *heap use-after-free*. Combined with the detection reports from ASAN, this suggests that these vulnerabilities are impacts of a same bug, though through different execution contexts. This intuition is validated by the fact that all these vulnerabilities are fixed by the same patch.

Illustration of limitations. In some cases, it is possible that vulnerabilities have greater capabilities than we were able to measure. For example, the size of the OOB write in *XML001* could be controlled implicitly via data as this vulnerability occurs during the parsing of a file, which mainly consists in string operations. Additionally, the repeated reads in *XML012* suggest that they could occur within a loop, of which the bounding condition could be controlled. Alas, detecting these parameters and tracking their constraints remains challenging for binary-level analysis, as discussed in Section 8.

Appendix X. Evaluation of Weak Interval Densities in Domains computed with Shrink and Split

In an ideal scenario, Shrink and Split yields exact domains of control. However, in practice, analysis may be interrupted or proving strong control on an interval may fail. This leads to only weak control being guaranteed on some intervals, while their actual density may change the interpretation of results.

To evaluate the impact of weak intervals on result interpretation, we computed the densities of those occurring in our analysis of benchmarks B1 and B2. We used ganak to count feasible values, then divided the result by the width of the interval to obtain its density. We then computed the average of these densities for each case.

Table 7: Weak interval densities on benchmarks B1 and B2

Program	S&S	S&SFB	Evaluation Impact*
uafubi	0.985	0.987	-
mul	0.5	no weak	-
mixdup	1.81e-05	1.81e-05	-
grub	5.7e-06	no weak	-
cve-2022-30790_2[1]	0.125	no weak	=
cve-2022-30790_2[2]	0.25	no weak	=
cve-2022-30790[3]	1	1	=
cve-2022-30790[4]	1	1	=
cve-2021-3246	timeout	timeout	timeout
heartbleed[5]	0.00117	6.04e-08	=
SSL002[6]	1	1	=
PDF019[7]	timeout	timeout	timeout
Average	0.486	0.664	

* impact on the score category from Table 5 (🙂 / 😐 / ☹). ↑: up a category, =: no change, ↓: down a category — [1]of_woff2 <5>, [2]woff2 <4>, [3]of_wsize <5>, [4]wlen <4>, [5]outgoing_packet <12> (leak), [6]write size, [7]read base address

As shown in Table 7, the densities of weak intervals span a wide range, with some almost empty and other completely full. As such, a systematic way of correctly handling them in practice seems out of reach. Nevertheless, in our experiments, these refined results would not change our score categories, solidifying our prior findings against the manual ground truth.

Appendix XI. Impact of Constraint Relaxation on Domains of Control

One way to improve the scalability of symbolic execution is to remove some constraints. This approach is called constraint relaxation or under-constrained symbolic execution [55] and may cause over-approximation. It may thus affect bug prioritization with our method if used.

To evaluate the impact of constraint relaxation on our method, we computed domains of control with S&SFB (100 maximum splits) based on the constraints from the vulnerable function only for each real-world vulnerability in benchmark B1 (except those where the vulnerable function was already the main analyzed function). We then computed their size ratio compared to the domains obtained in our mainline experiments.

Table 8: Domain size ratios with and without relaxation for benchmark B1 (S&SFB, 100 max. splits)

Bug	Eval. Impact*	Sink	Domain Ratio
heartbleed	=	payload_size	1
cve-2024-43700	=	Detection_0_CFH_Ret_byte_(7-0) <7-0>	1.01
cve-2024-41881	=	Detection_0_CFH_Ret_byte_(7-0) <7-0>	1
cve-2021-3246	=	of_size <1>	9.39e+13
		blocksize <0>	3.3e+04
cve-2020-14393	=	Detection_0_CFH_Jump_byte_(7-0) <7-0>	1
cve-2019-19307	=	len <1>	1
		end <0>	1
cve-2019-14202	=	of_wsize <1>	6.67e+04
		len <0>	6.61e+04
cve-2019-14192	=	of_wsize <1>	6.67e+04
		len <0>	6.61e+04

*impact on the score category from Table 5 (🙂 / 😐 / ☹️). ↑ up a category, = no change, ↓ down a category

As expected, Table 8 shows that the domains of control with relaxed constraints are always equal or larger than those for the exact constraints (over-approximation). More precisely, the domains are very similar in half the cases (6 out of 12), but we observe very significant difference on the other half (6 out of 12). Yet, interestingly, our score categorization from Table 5 would however not be affected by this approximation, as the largest ratios occur here for vulnerabilities with already high capabilities. In terms of performance, symbolic execution runtime ranges from unaffected to substantially reduced (28x faster), yet the overall analysis runtime is only marginally affected (2x faster at best), since tracing takes the most time due to the large difference between the size of the whole trace and what is symbolically followed (see Appendix VI).

Appendix XII. Impact of Input Concretization on Domains of Control

Another way of improving the scalability of symbolic execution is to concretize part of the symbolic inputs, in order to reduce the complexity of the constraints and the number of symbolic state updates [20, 30]. This approach may cause under-approximation and thus affect bug prioritization.

To evaluate the impact of input concretization on our method, we computed domains of control with S&SFB (100 maximum splits) for each real-world vulnerability from benchmark B1 with an arbitrary part of the input concretized. We then computed the size ratio compared to the domains obtained in our mainline experiments.

Table 9: Domain size ratios with and without partial input concretization for benchmark B1 (S&SFB, 100 max. splits)

Bug	Eval. Impact*	Sink	Domain Ratio
cve-2024-43700	=	Detection_0_CFH_Ret_byte_(7-4) <7-4>	0.00403
		Detection_0_CFH_Ret_byte_(3-0) <3-0>	1
cve-2024-41881	=	Detection_0_CFH_Ret_byte_(7-4) <7-4>	0.00394
		Detection_0_CFH_Ret_byte_(3-0) <3-0>	1
cve-2023-37837	=	read_off <0>	0.00391
cve-2022-30790_2	=	woff <8>	1
		wlen <6>	1
		of_woff2 <5>	1
		woff2 <4>	1
		woff <2>	1
		wlen <0>	1
cve-2022-30790	↓	of_woff <7>	0**
		woff <6>	0.995
		of_wsize <5>	0**
		wlen <4>	6.11e-05
		woff <2>	1
		wlen <0>	6.11e-05
cve-2022-30552	=	of_woff <3>	0
		woff <2>	0.000488
		of_wsize <1>	0.000489
		wlen <0>	0.00104
cve-2021-3246	↓	of_size <1>	0.00288
		blocksize <0>	0.00393
cve-2021-26567	=	Detection_0_CFH_Ret_byte_(7-4) <7-4>	0.00392
		Detection_0_CFH_Ret_byte_(3-0) <3-0>	1
cve-2020-14393	↓	Detection_0_CFH_Jump_byte_7 <7>	1
		Detection_0_CFH_Jump_byte_6 <6>	1
		Detection_0_CFH_Jump_byte_5 <5>	0.00781
		Detection_0_CFH_Jump_byte_4 <4>	0.00391
		Detection_0_CFH_Jump_byte_3 <3>	0.00391
		Detection_0_CFH_Jump_byte_2 <2>	0.00391
		Detection_0_CFH_Jump_byte_1 <1>	0.5
		Detection_0_CFH_Jump_byte_0 <0>	1
cve-2019-19307	=	len <1>	0.545
		end <0>	3.81e-06
cve-2019-14202	↓	of_wsize <1>	1.55e-05
		len <0>	1.54e-05
cve-2019-14192	↓	of_wsize <1>	1.55e-05
		len <0>	1.54e-05

*impact on the score category from Table 5 (🙂 / 😐 / ☹️). ↑: up a category, =: no change, ↓: down a category — **here the feasible values do not even allow OOBs

As shown in Table 9, and as expected, the resulting partially concretized domains of control are always equal or smaller than the exact ones (under-approximation). More precisely, the domains are very similar (> 0.99) in 14 out of 39 cases and significantly different (< 0.1) in 22. For 5 out of 12 vulnerabilities, our assessment from Table 5 would have been affected, with lower scores leading to a lower threat classification. Similarly to relaxation, symbolic execution can be faster (up to 83x), although overall performance gains are again small due to the tracing runtime.