A theoretical system for automatic generation of overlapping x86-64 binaries

Sebastiaan Peters

I. Introduction

Programs written for the x86-64 architecture have many ways of being represented, both on a semantic level (which sequences of instructions are chosen to achieve a desired effect), as well as on a syntactic level (which encodings are chosen to represent an instruction).
The architecture specifies that it should be able to start execution on different offsets irrespective. Meaning that we can have execution that the same byte sequence can be read from different starting points, and obtain different unique sets of instructions as a result. Binaries which exhibit that *overlapping* behaviour typically are not seen through using mainstream tooling for program compilation/assembly.

In this work we investigate whether a non-structured overlapping of x86 code exists on a greater scale without resorting to a fixed underlying structure. To the best of our knowledge this would be the first work providing a system capable of making convincing argument for or against the existence of pieces of x86 code that would be loosely overlapped in the range of 100 or more instructions.

We approach this by going through going a large amount of representations for the same program, and observe on how successfully we are in overlapping. We measure success in two ways, one being the relative success of how many transformations we need to apply to gain a certain amount of overlap in bytes, and the other being how much of a program we can overlap. Such that even if we could not achieve the goal of overlapping specific programs, we could still gain some insight in the generalizability of our techniques to other programs.
While we can't go through all the infinitely many representations we argue that we can determine practically on the basis of being able to exhaustively test all the syntactic representations of a program, and explore the space of programs yielded by applying a permutation of all current day compiler optimizations.
We believe that the diversity of current day optimizations should be enough such that a new optimization would not diversify massively on a synatic level.
In particular we will show that if we have a database of overlapping programs which happen to have no side-effect when executed, that this is sufficient to prove it is possible to arbitrarily overlap programs, as these are able to be inserted in any other program at any point (including from this same class). Effectively being able to patch up a sequence which wouldn't be able to overlap by changing the underlying byte sequence without having an effect on the program. Meaning that if we find a small successes here, we can apply these successes to create even more diverse set no side-effect programs and continue doing this recursively.

With this, we hope to present a solid indication into the feasibility of non-structured overlapping binaries. With the goal to enable informed decisions on whether people working in areas where this could be applied to should concern themselves with the possibility or not.

II. Background

Digital computers are machines that interpret digital signals and change it's state based on them. What actions these computers can take and how signals map to actions is captured in a document named the *Instruction Set Architecture* (ISA). An action implemented by the machine is called an *instruction*. The digital signal is typically a sequence of bytes $\mathcal{B} = (b_1, b_2, ..., b_n)$. If there exists a mapping between $\mathcal{B}$ and an instruction, we call $\mathcal{B}$ an *encoding* for the instruction. In the x86-64 ISA created by AMD there exists multiple encodings for the same instruction. Note that it technically does not makes sense to talk about the length of an instruction, due to that an instruction can have multiple encodings with different lengths.

Successfully converting $\mathcal{B}$ into an instruction is called *decoding* and is done by a decoder. Decoders can exist both as physical hardware inside a CPU, or as software form. The length of an encoding is equal to the bytes consumed from the signal. For an encoding e this is denoted as $|e|$.

All programs eventually reduce down to a long sequence of encodings concatenated at after each other with possible padding in between. We assume that $\mathcal{B}$ represents some program in this text.

With $\mathcal{B}_o$ we denote the ***encoding*** *located at* the *offset* o inside binary $\mathcal{B}$

Equality between two encodings is byte-wise, thus if the byte sequence is the same, but located at different offsets, then they are still considered equal.

One last important concept is $Enc(\mathcal{P})$. This represents one of the possible sequences of encodings of program text that would successfully execute program $\mathcal{P}$ if it were to be executed. We also assume that the sequences provided by this are minimal in the sense that removing any encoding

would break the correctness of $\mathcal{P}$. We include this notion because a target binary may contain an arbitrary bytes as long as they do not get executed.

A. Decoding streams

A decoder when given a binary $\mathcal{B}$ starts decoding at a given offset o. After having decoded an instruction encoding e, it continues on with decoding from the next byte onwards located at the offset $o + |e|$. The sequence of encodings obtained by continuing this is called a *Decoding Stream*. We call a stream ***valid*** over a sequence if from the starting point onwards, we always are able to return an instruction. If for some reason we encounter a sequence which are can not be converted back to an item in the ISA, then we call it ***invalid***. Typically $\mathcal{B}$ contains multiple Decoding Streams which are not a subsequence of each other. Take figure 1 as an example. Here we can see that if the decoding stream would start at offset 0, we would get a sequence starting with the "add bl, al" instruction followed by a jump, where as if we would start at offset 1 we would get the "ret" instruction. Note that at no point if we did start at offset 0 we would decode a ret instruction.

$\mathcal{B}$ = 00 C3 EB C3		
Offset in $\mathcal{B}$	encoding	instruction
0:	*00 C3*	add bl, al
1:	*C3*	ret
2:	EB C3	jump 0xfffffffffffffc5
3:	C3	ret

Fig. 1: C3 used inside two different encodings

B. Different decoding stream consumers

¡— redo this since it's kind of shitty/ we can beter split it up in saying - there are two main camps decoding linear sweep rec decent

linear sweep has disassemblers rec decent has disassemblers, but most notably models the cpu

The context in which a decoder is used can vary widely. But we want to highlight two pieces of technology using decoding streams:

- A disassembler
- An Execution Unit

1) Disassemblers: While there are a large variety of disassemblers and how they work (cite SoK about disassembly and hopefully extract decoding stream from there), we will focus our attention on linear sweepers. A linear sweeper disassemblers are tools that typically follow a decoding stream from a single offset, and try to pretty-print the resulting decoding stream as is.

2) Execution units: A key feature of the execution unit is that it does not necessarily following the decoding stream it has started of, but rather saves the value of the next offset to start decoding at in a special register called the Instruction Pointer (IP). Meaning that if an instruction got decoded and subsequently executed which would alter the IP, then the decoding stream would be altered as well.

C. Divergence techniques

Due to the fact that one system guides how it will decode a stream based on what it instructions its decodes, while the other one doesn't, implies that there exists multiple ways in which we can cause divergence in the output between systems. Applying any of the divergence techniques listed below results in that we have two *divergent decoding streams*. This means we can have two decoding streams DS_e and DS_d who are given the same offset, which yield different sequences of instructions encodings. Here we assume DS_e represents an decoding stream in an execution unit and DS_d an decoding stream based on a linear sweeper disassembler.

The case where an instruction would alter the IP of DS_e into the start of an encoding in DS_d is called *aligned divergence*. It is also possible the IP gets to a byte which is not the first byte of an encoding in DS_d. This is called *unaligned divergence*. Typically when using a linear sweeper we do not see aligned divergence as a problem. Since program analysis is done by an operator (be it a tool or a human) and all the pieces of how the program behaves should be visible. Rather this paper will focus on the case of unaligned divergence.

We assume that both decoding streams after unaligned divergence continue to be valid indefinitely. This is an assumption that typically does not hold in reality, but we make this assumption for clarity. After two streams have diverged, they may converge back into the same stream directly in the next instruction or after any number of instructions.

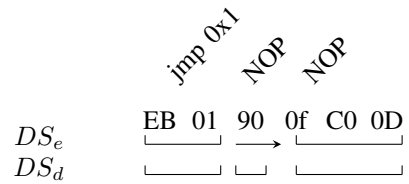
Below are two different techniques for creating unaligned divergence:

1) Jump based divergence: There are instructions which allow the instruction pointer to be changed to an arbtirary value. If an instruction i has an argument which is an offset that would at a non-starting byte of an encoding read by a linear sweeper, we have that the linear sweeper would continue to the next byte after i. While an execution unit would cause the next encoding to be read at an unaligned offset. Causing unaligned divergence.

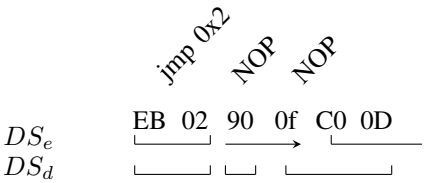
2) Divergence based on implementation issues: Another method of causing divergence is using erroneous implementation combined with error recovery. For instance the tool GNU Objdump chooses to error recover when it is unable to decode an instruction. If decoding at offset o would not yield a valid instruction, it would continue by trying to decode at offset $o + 1$. When objdump does not accept a valid instruction i, it causes the scenario that for a decoder which would accept i, the decoding stream would diverge by the difference i and the error recovered offset.

3) Example of divergence: Figure 2 illustrates aligned vs unaligned divergences. The figure contains two different programs each consisting of 4 parts. The programs differ in that the second program will jump one byte further. In the middle is a sequence of bytes representing $\mathcal{B}$ in this example, To the top we have it's corresponding disassembly. At the

bottom we have DS_e and DS_d with a list of ⌴ shapes under the bytes. These denote the grouping of which bytes will be seen together as a single encoding for an instruction The arrow represent which bytes will get skipped by the decoder. Notice that in the case of aligned divergence, the sequence of encodings for DS_e is a sub-sequence of DS_d. While in the unaligned case we would two different sequences due to the C0 0D bytes being part of the beginning of an instruction in DS_e, while this not being the case for DS_d.



(a) aligned divergence



(b) unaligned divergence

Fig. 2: Aligned vs unaligned divergence

D. Divergent streams

Let us consider two decoding streams which are unaligned divergent DS_1 and DS_2. By the definition of being diverging there must exist two encodings $e_1 = (b_1, ..., b_n) \in DS_1$ and $e_2 = (c_1, ..., c_m) \in DS_2$ such that either the starting offset of e_1 in $\mathcal{B}$ is before the offset of e_2 and that offset in e_2 is at the offset of $e_1 + x$ with $x \in [1, 2, .., |e_1| - 1]$, or the offset of e_2 must be before e_1 and e_1 must be at $e_2 + y$ with $y \in [1, 2, ..., |e_2| - 1]$. Let us pick DS_1 and DS_2 such that the starting offset of e_1 in $\mathcal{B}$ is smaller than e_2.

By our choice we have that there exists some $1 < j \leq k$ such that $(b_j, ..., b_k) = (c_1, ..., c_m)$ with $m \leq n$. Or rather put, the bytes of the encoding for e_2 overlaps with some bytes past the first byte of e_1. There are two different cases. It might be that all bytes of e_2 are contained in e_1, or they are only partially contained. These are captured in the following two notions:

1) Fully embedded: We say that e_2 is ***fully embedded***($\subseteq$) in e_1 in a given binary $\mathcal{B}$ when $(b_j, ..., b_k) = (c_1, ..., c_m)$ for some $1 < j \leq k \leq n$ and $e_1 = \mathcal{B}_p$ and $e_2 = \mathcal{B}_{p+j}$. Thus an encoding is embedded in another if the entire encoding is present inside that of another for inside the given binary. If e_2 starts at the first byte of e_1 (or $j = 1$) then they are equal and hence are we not include it in our definition.

2) Cross embedding/crossing: We say that e_1 is ***crossing***($\times$) e_2 in a binary $\mathcal{B}$ if and only if for $e_1 = (b_1, ..., b_i, ..., b_j)$ and $e_2 = (c_1, ..., c_n, ..., c_m)$ we have that

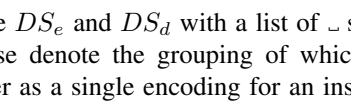


Fig. 3: Example of overlapping encodings: $e_1 = (XX, YY, ZZ, AA, BB)$, $e_2 = (ZZ, AA, BB) : e_2 \subseteq e_1$

$(b_i, ..., b_j) = (c_1, ..., c_n)$ with $1 < i < j$ and $1 \leq n < m$ and $e_1 = \mathcal{B}_p \wedge e_2 = \mathcal{B}_{p+i}$.

This definition states that $e_1 \times e_2$ iff both encodings have some byte in common, while each also has an byte that is not contained in the other.

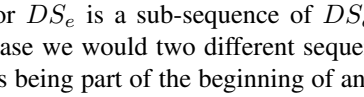


Fig. 4: Example of overlapping encodings: $e_1 = (XX, YY), e_3 = (ZZ), e_4 = (AA, BB)$ is $e_2 = (YY, ZZ, AA) : e_1 \times e_2 \wedge e_2 \times e_4$

Observe that both definitions are ***not*** symmetric. It also follows that if e_2 is fully embedded in e_1, we by the assumption that DS_2 is valid indefinitely have an encoding e_3 directly following e_2. The case that e_3 is also fully embedded into e_1 can only be repeated at most for $|e| - 1$ times. Hence if we wish to continue the divergence for more than only for e_1, then either we need an encoding that crosses e_1, or we need to reapply a unaligned divergence technique. Conversely, it is impossible to have two unaligned streams converge if they only contain crossing encodings.

3) Overlapping: We call any two encodings e_1 and e_2 ***overlapping*** if one is embedded in the other, or they cross each other. For two decoding streams to be overlapping over the range over $[i, j]$ we require that all encodings in offsets between i and j inside $\mathcal{B}$ to be overlapping for any two decoding streams starting possibly different offsets.
When we say ***to overlap*** $\mathcal{P}$, we mean to alter $\mathcal{P}$ into an equivalent program w.r.t execution but has one or more decoding stream inside $\mathcal{B}$ for which it overlaps

E. Difficulty in overlapping

To create an binary which contains an overlapping binary for an extensive range is a hard thing to do. There are two major underlying reasons for this.

1) A fragile ISA: The first is that the x86-64 encoding scheme is relatively fragile. Meaning that if we pick bytes at random, the chance of them forming a valid instruction is relatively low. Although no exact numbers are known we can make a quick estimation. C. Bölük (2021, Haruspex https://haruspex.can.ac/chip/6850k/grid/) showed that around $\frac{3}{16}$ of the possible values for the first byte would result instantly in an invalid encoding. If we we're to follow a geometric distribution this would imply 95%+ of sequences consisting of randomly selected bytes would yield an invalid instruction within 5 instructions.
This is relevant because of the requirement that without continuously re-applying divergence techniques, we require crossing encodings to exhibit continued unaligned

divergence. Implying that in a process of selecting encodings for instructions used to represent a program to be encoded, we have that without regard for the encoding in another decoding stream, bytes will be selected. These can be seen as random choices hence giving us a scenario in which the above result severely hinders our chance of successfully generating overlapping instructions. If we were to give thought to a choice of encoding would impact another decoding stream, we need to apply program transformations which are not typically limited and not trivial.

2) Automatic convergences: The second reason is that due to something called the Kruskal Count(James Grime, Kruskal's Count, http://singingbanana.com/Kruskal.pdf). If we have sequence s where $s \in \mathcal{N}$. Then we can start the following process: First we pick an arbitrary index i to start at. Next we look at the value located at index i and jump to index $i + s$. It turns out that if you have two people following this process, there is a very high likely-hood of converging before too long. This process is also identical to what an decoding stream is, with the change of that instead of some arbitrary s we have that s represents the size of the encoding at index i inside $\mathcal{B}$.
The Kruskal principle has 2 parameters, the average step size x and the number of entries in the sequence N. The chance of converging is expressed by:

$$P(converging) = 1 - \left(\frac{x^2 - 1}{x^2}\right)^N$$

.
Considering that the average encoding length inside a program is 2.47 bytes. (cite: (2020, Evaluating Security of Executable Steganography for Digital Software Watermarking)). Meaning that we have a 95% to converge after 15 instructions.

F. Prior work

Due to the converging nature of x86, current techniques for creating a long divergent stream has been focused in two areas:

- Fully embed $e \in Enc(P)$ into a cover program
- Junk insertion

1) Fully embed $e \in Enc(P)$ into a cover program :
This technique is presented in by C Jämthagen (2013, A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries), W Mahoney(2018, Leave it to the weaver) and JA Mullins et all(2020, Evaluating Security of Executable Steganography for Digital Software Watermarking).
In x86 we have instructions can have immediate operands. These can be seen as hardcoded arguments to an instruction. It is possible let these take on any value, so also $e \in Enc(P)$. Which automatically would yield e to be fully embedded. However, the space in which code can be embedded is only a few bytes. Meaning that if we want to embed a program larger then that, there needs to be a way to keep flow control going from one embedded instruction to the next.

The first option would to reserve bytes at the end of the operand to jump to the next instruction in the sequence. This method allows for a great spread inside the binary the next to be executed instruction resides. However a jump requires at least 2 bytes, while operands typically have at most 4 bytes (with the exception of a single rarely by a compiler generated mov-instruction). Even though the typically length of an instruction is above 2 (cite: leave it to weaver v2). Meaning that we would be unable to express all desired programs.

The second option is to reserve the last byte, together with carefully chosen encoding for which it gets embedded onto, to provide an as side-effectless instruction so that even though an instruction does get added to the instruction stream, which does not disrupt it.

Both of these options imply that the cover program will needs to contain instructions of $4+$ byte operands or very specific nop instructions on approximately the order of the size of the overlapped program. For both approaches Jämthagen and Mullins showed that is possible to create a detector based on frequency anomalies of these instructions. Due to these inherent limitations on x86-64, we shall focus on overlapping binaries in which $e \in Enc(\mathcal{P})$ are crossing.

2) Junk insertion: C. Linn(2003, Obfuscation of executable code to improve resistance to static disassembly) proposed the notion of junk insertion, which means that for each $e = (b_1, ..., b_n) \in Enc(P)$ a sequence of bytes $c_1, ..., c_n$ would be brute-forced and be placed before e to form a sequence $(c_1, ..., c_n, b_1, ..., b_n)$ such that a decoding stream starting at c_1 and b_1 would be divergent, but would converge to the same point.
This allowed to have $e \in Enc(\mathcal{P})$ to be both fully embedded as well as crossing. However the quickly converging nature of x86 meant that not all the entirety of $Enc(P)$ could be hidden if it was more than a few instructions long.

D Andriesse(2014, Instruction-level steganography for covert trigger-based malware) improved on this notion. If only x encodings could be hidden through junk insertion, he would re-apply junk insertion to the $x + 1$th encoding allowing to fully overlap a given $\mathcal{P}$. To deal with the extra junk inserted in-between instructions he would add jump instructions fixing the flow control. Code still converges before the newly inserted jump instructions and hence the jumps would be visible.
The downside to this is that due to the fact that we need re-apply junk insertion in the first place, namely that the hidden sequences typically are of short length, we have that we need many re-applications and hence jumps that fix the flow control. Therefore it would be easily detected and still would leave us in an similar spot compared to the approach of only using full embedding encoding.

¡– should also mention that parralax did things a little bit

betteer

G. Program diversification

Program diversification aims to generate multiple variants of a program while preserving it's semantic meaning, this can be done to improve the end binary of a program in any metric, notably performance or susceptibility to exploits.

Due to the limited success of C. Linn, program diversification was applied to the program before junk insertion. This was done through nop/dead-code insertion between encodings which were tried to be made overlapping. However this was deemed unsuccessful in raising the percentage of encodings that could would be overlapped through junk insertion.

< meta comment: Linn mentioned that he did program diversification by summing up 3-4 techniques and that it did not made a significant impact. He wrote this offhandedly inside the related work section. This makes me not sure how to represent this since I can't clearly interpret the extend of his results >

However the included program diversification techniques notably did not include semantic preserving transformations on the input program it self, as well as the possibility of using equivalent encodings inside the ISA.

H. Applications

1) Divergent streams as obfuscation tool: Due to nearly all binary analysis being on top of a disassembler in some form, being able to create a divergence between what would be executed on the CPU compared to the disassembler would be a powerful tool to hide code. With divergence it could be that a hypothetical anti-virus scanner would not detect malware since it's what the tool would see would differ from the behaviour of the malware. Therefore it is of great interest to be able to create binaries which are fully overlapping for a program $\mathcal{P}$.

It turns out that generating overlapping code is quite hard. In practice it seems that most assembler/compiler generated binaries do not diverge for long. C. Linn et all in "Obfuscation of executable code to improve resistance to static disassembly" showed that only 1% of code divergence build with typical tooling.

2) Program Steganography and defensive ROP: In the context of Return Orientented Programming there exists the notion of unintended gadgets. Gadgets are small pieces of code preceding a call or similar control flow transfer instructions meant to chain programs with. It is possible that the start of these gadgets lay at the offset not present in a function created during compilation for the host program. This is actually analogous to two linear disassembler decoding streams overlapping.

ROPSteg(K Lu et all, 2014) explored this notion to hide parts of a programs these unintended gadgets for what they call Program Steganography. They experienced a sharp drop off in the chance to overlap pieces of code and resorted to mostly splitting up code to very small(2-13) byte sequences. This in particular makes it harder to make unintended gadgets create call-preceeded gadgets (ROP is Still Dangerous: Breaking Modern Defenses, 2014, this has a nice list to cite).

3) Code integrety: For Parralax(D Andriesse et al, 2015) Andriesse showed how overlapping code segments can be used to improve code integrity through overlapping code of a program with code that is part of a verification function. This way integrity verification functions would certainly fail if the underlying code was tampert with. This is in contrast to ROPSteg which only embedded code part of the desired program directly.

4) Performance: Depending on how successful overlapping techniques are, it might be possible to vastly decrease code size. Consider any instruction sequence $\mathcal{I}$ that is encoding into a sequence of bytes $\mathcal{B}$. It could be possible to overlap the latter half of $\mathcal{I}$ with the first half. Meaning that in theory we reduce the size of $\mathcal{B}$ by half in the perfect scenario.

III. Related work

To be filled in later

IV. Our research

While we are able to generate overlapping binaries with crossing encodings for an arbitrary length (jalhmtagen), unfortunately the crossing encodings are always pre-determined and might alter our input program in some way. This has as side effect that all encodings used to represent the semantics of our input program needs to be completely fully embedded. This leads to issues where the overlapped binary would not be diverse enough to be used in contexts of obfuscation. Conversely, current approaches which allow for encoding denoting semantics of the input program to be crossing seem unsuccessful in creating large sequences that overlap.

We wish to explore if we can remove these limitation and automatically generate overlapping binaries for real world code where encodings representing the semantics of an input program are crossing and are also on the order of 100-1000 instructions long through the enumeration of possible representations of our input program.

While it is known that conventional build tooling does not produce overlapping binaries, these tools typically contain a rich set of possible program transformations; but bundle the application of these into a select few of optimization settings. We believe program transformations found in compilers and program diversifiers, in combination with redundancy in the ISA could provide a mechanism to automate the generation of larger overlapping binaries.
It follows necessarily from creating overlapping binaries that we will have two decoding streams. We only put the

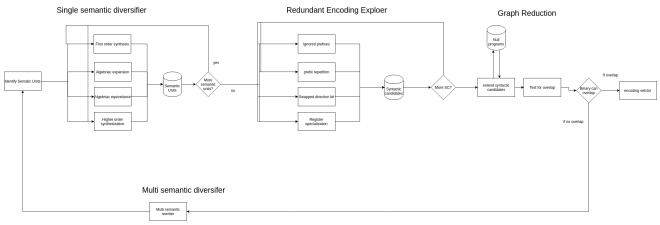


Fig. 5: Overview of the system, the figure will be improved in the next version

requirement that one of these should exhibit the semantics of our input program and put no constrains on the other stream other than that it must contain crossing encodings. We leave the generation of where we can specify semantics in more than 1 streams up for future work.

The main hurdle in this question is that for any given program, we have an infinite amount of different ways to represent them. Each of which could be an overlapping binary. Even if we do not set the requirement of a concrete mathematical proof that the existence for a certain program to overlap is impossible, we did not find arguments described in literature that aims to be convincing on an intuitive level either.
Therefore our paper contributes a theoretical system aiming to:

1) Automate the generation of overlapping binaries
2) Provide a convincing argument that in the absences of results of our system it would be impossible to overlap a "real-world" program depending on how diverse the program transformations manifest on an encoding level

We recognize that the notion of "real-world" is arbitrary. However we still see value in this claim. We hope to enable informed decisions on whether expert which possibly would have to deal with this class find it worthwhile to consider in their domain.

V. Methodology

2 PHASES, bootstrapping/initialization phase, - generate null running phase -

- all of the blocks better highlighted in this text

We shall investigate our idea by creating a system that intelligently explores the space of all equivalent programs through two sets of program diversifiers. We take an input program $\mathcal{P}$, and first run it through the first set of program divserifiers that concerns it self with semantics the size of individual instructions (Single Semantic Diversifier). For each set of these diversifications we produce a complete set of all the different ways we can encode these instructions in x86 (Redundant Encoding Emitter). Subsequently we test each potential pair of encodings for if could aid in overlapping (Graph reduction).

If it it turns out that we can fully overlap a program run our results through the encoding selector, which will emit the final output. It walks through the possible options for creating a overlapping binary and will subsequently pick out the best one according to some criteria or at at random. Otherwise we will mark the areas which don't overlap as problematic and run a set of more sophisticated diversifiers (Multi Semantic Diversifier) until all of these problematic instructions have been changed in some fashion. Once all the problematic areas are gone, we use the new program and re-run the system from the start on the new altered program.

A. Bootstrapping

For our system we shall make a key observation on how we test for overlapping-ness between two encodings e_1 and e_2. Namely that even if e_1 and e_2 would not overlap in anyway, that there could exists a series of encodings $(e_i, ..., e_j)$ such that the execution of e_i to e_j would not yield different semantics than without it's inclusion. But would still cause the chain of $e_1, e_i, ..., e_j, e_2$ to overlap (requiring that $(e_i, .., e_j)$ also internally overlap. Let the space of all programs which do not exhibit side-effects for the program be called *Null Space* or $\mathcal{N}$
Hence if we would found a set for $\mathcal{N}$ such that they would fill up all the ranges for all combinations for possible starting and ending bytes that would fix up non-overlapping programs, we would automatically prove of being able to overlap arbitrary binaries. These side-effect less instruction sequences are were named ineffective instructions in ROPSteg.
As a result, we diverge in two major ways compared to previously literature:

- We would like to include larger binary sequences(15+ bytes) if need be
- We recursively apply our findings to increase find more ineffective instructions

To achieve this, we first bootstrap $\mathcal{N}$ requiring a set of programs which we apriori know that they shouldn't exhibit side effects. By running these a couple times through our main system, we hope to gain a larger set of side-effectless programs.

Once we have done this a few times, we can apply the same principle to a larger set of programs which we know have no side-effects, and constantly reapply our findings to the same set until we have hopefully we have covered the entire space of bytes that could potentially be fixed up.

Secondly, we hope that through recursively applying newly found ineffective instructions to previously not-overlapping ineffective instructions will cause for a rapid explosion in ineffective instructions available to us.

VI. Semantics and syntactic candidates

Meta note: The following two concepts are used pretty heavily everywhere from now on, but felt like they are too

specific to what we're building so I only put them directly above boxes in our system.

A. Semantic Unit

The first concept is that of ***semantic units***. A semantic unit S describes the alteration of state in the machine for a given sequence of instructions within the ISA. Or otherwise put as the concrete behaviour of a sequence of instructions. Note that while it is possible to have more complex Semantic units, we suspect that most of our analysis/transformations on an semantic units are better suited the smaller the scale.

sugar for the semantics that specify each instruction. We introduce this notion because of the redundancy in the ISA and ability to synthesize instructions from each other.
Consider the following example: the (near) ret instruction is defined to be a $pop\ \ rip, jmp\ \ rip$. As a direct consequence, any program in which ret gets substituted with $pop\ rip,\ \ jmp\ rip$ should execute the same semantics. Hence we consider them as the same semantic unit.
Because semantic units represent sequences of instructions, we can represent our input program as a sequence of semantic units.

B. Syntactic candidates

We define a ***syntactic candidate*** as sequence of encodings that strictly implement for a corresponding semantic unit fully, and no more. There exists a one-to-many relationship between a semantic candidate and syntactic candidate. We include this notion because it might be that a syntactic candidate could consist of multiple instructions. Causing them also to be possibly longer than 15 bytes.

VII. Single semantic diversifier

Due to our testing methods only being able to test for overlap between two directly adjacent semantic candidates, therefore we will expand all possibilities for implementing the same single semantic unit. We do this first on a strictly instruction level, and convert these into encodings in the next part.
To apply transformations, we take the original instruction given in the program and apply transformations on it. After every set of extra options we apply these transformations again on the new set. Due to some generation methods being able to expand infinitely, we implementation an artificial limit to would cause this step to halt after a pre-determined amount of iterations/time.

We identify four different categories of Single semantic diversification. While experimental data would likely be required to verify, we suspect that transformations in these categories differ widely in how they alter underlying bytes compared to those in other categories.

1) First order synthesizing: The first transformation category is that of First order synthesization. A semantic unit can be created by a composition of other instructions within the ISA, but without the synthesized instructions first forming an abstract computation machine as described in the section about Higher order synthesis.

For instance a the near return instruction is defined to be equivalent to first execute the same semantics as defined in $poprip$ and then by $jmprip$. Due to this we can state that their semantics are equivalent and the latter synthesizes the first. Hence in our final program it should not matter which version is picked.

2) Arthemitical equivalence: Mathematical operations typically are equal to each other for certain input sets. For instance multiply by 2 is equal to adding a value twice.

3) Arthemitical splitser (linear combinations): Most arithmetic are linear. Hence we can represent any addition and multiplication as a linear combination of these operations meaning. Note that special care should be taken to preserve the semantics of the program w.r.t flags

4) Higher order synthesis: x86-64 has many possible ways of implementing abstract computing machines which in turn can simulate instructions which are present inside the ISA it self. A famous example of this is that mov is Turing Complete, and hence can simulate all compute mechanisms of the ISA. x86 exhibits a large range of possible instruction combinations that can implement a Turing Machine, which by definition is able to simulate any other Turing Machine. Because of this we can perform have recursive implementations of turing machines implementing each other. Causing any calculation able to be performed in an infinite possible of ways.
While this means that a single instruction can be expressed in an infinite amount of ways, we might still be able to fully reason about possibility of overlapping instructions for this category if we can make statements about the underlying implementations of these abstracter computation machines.

VIII. Redundant Encoding Explorer

For every instruction in the x86-64 architecture there exists a multitude of encodings expressing them. Most of these stem from using prefixes or from trying to express instructions which are symmetric in their arguments.

This part of the system takes all the instruction sequences that would implement semantic units and converts them into encodings. Thus automatically transforming these sequences into syntactic candidates. Afterwards, we will feed these encodings into sequence of expander units which will emit extra encodings based on which rules apply to that specific encoding. Notice that the expansions methods below can be chained together.
This space of looking for redundant encodings is little researched. Primarily because a lot of option perform strictly worse in nearly all traditional metrics like encoding size. As far as the authors are aware, irasm

(http://github.com/xlogicx/irasm) is the only tool that has the intention of exploring these more alternate encodings.
Giving a comprehensive overview would be a complex endeavour considering

- The "legacy" encoding scheme of x86-64 is widly considered to be the most complex of all in current day use
- The numerous extensions with some bringing in their own encoding scheme (VEX/XOP)
- The effect of prefixes can be dependent on which instruction they are being applied
- The use of recurring prefixes is in most technically undefined behaviour

Below is our list of possible redundant ways to encode instructions.

Prefixes:
Prefixes can cause redundancy through being ignored, repetition, or through canceling each other out.
In x64 there are two cases where prefixes are ignored. The first being that the REX prefix is ignored when it does not preceed the opcode byte directly. Another case is by prefixing an instruction with the CS, SS, DS or ES prefix. It also allows for repeating prefixes, For instance it is valid to have the $66h$ prefixes repeated twice. The repetition does not need to directly follow each other, thus sequences of the form (p_1, p_2, p_1, p_2) where p_1 and p_2 would be two different encodings for prefixes would also be allowed.
x86 has a 4 group of prefixes for their core instructions. It is undefined behaviour when more than one prefix of a group is included in the instruction. Typically processors opt to ignore all but the last of each group. Although more research needs to be done on how undefined behaviour manifests on different processors.

1) direction bits: For most "core" x64 instructions we have that the second least significant bit of the operand byte dictates what the order of operands is are specified in. Hence if an instruction is symmetric we gain two different methods of encoding the same instruction due to order not mattering anymore.

2) lowering register: $x64$ allows to modify the operand width on an instruction. Meaning that we can write a 32-bit value into a $64-bit$ register by specifying eax as the target register instead of rax. Writing a $32-bit$ value into a $64-bit$ register (or a 16 bit value into either 32/64 bit register) zero's out all higher bits. As long as the value written is has a zero sign (since all higher bits are zero'd) and value is less than typically represented in $32-bits$, we can change an $64-bit$ write into a $32-bit$ write.

A. register specialization

For particular the $[r|e|a]ax$ register we have that there exists for some instructions i there exists two opcodes. One opcodes for specifying i can take any register as argument, and one specialized opcode that denotes i with $[r|e|a|]ax$ as argument.

Since $[r|e|x]ax$ can still be explicitly stated under the first opcode as argument, we now have multiple encodings.

1) VEX duplicate prefixes: To be filled in later

IX. Testing for overlap

Because our system is aimed at performing complicated program mutations, combined with the fragile nature of overlappingness, we wish to test not just for if the entirety of the program is overlapping, but also to locate which parts are overlapping and which are not. This should be able to prevent the program rewriter to unnecessarily alter overlapping parts maintaining a sense of progression over each step of our system.

The input to this system is a sequence of semantic units representing the program each accompanied with a set of potential syntactic candidates. Our goal is to find a sequence that:

- Contains a single syntactic candidate for every semantic unit in the same order as the semantic units
- Have an divergent decoding stream from a starting point anywhere within the first syntactic candidate, and converges on the last encoding of the last syntactic candidate

Such a sequence should automatically express the desired semantics of the input program by virtue of being a chain of syntactic candidates that was in the same order as every given semantic. Together with having a divergent decoding stream it would satisfy as a proper overlapping program for our input program.
To first find it's existence, we first test whether between any combination of selection in neightbours from syntactic canididates cause an overlapping relationship. We test more than just the adjacent candidates considering those can be fully embedded and hence don't necessarily complete an instruction. Afterwards we pick a single syntactic candidate that would have an overlapping relation for each semantic unit. This implies that we concatenate our choices for each semantic unit into a single binary sequence.

However concatenation of syntactic candidates does not necessarily preserve the validity of a decoding stream even if the candidates overlap.
Consider the following example:

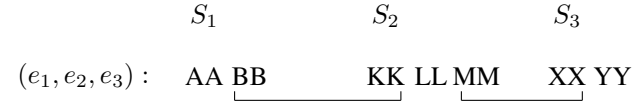


Fig. 6: Example of how concatenation does not preserve stream validity

In Figure 6 one syntactic candidate for all three semantics. They all are crossing with each other, however if we would assume that no valid encoding could start with LL, we wouldn't have a valid stream.

A. Syntactic candidate start and ends

To maintain validity after concatenation we define the ***extendend syntactic candidate***. This extended syntactic candidate esc includes five extra properties:

- start
- internally consistent
- converges
- end
- is fully embedded

Start:
Assume we have an encoding e and we have $e \times esc$. This means that some bytes at the beginning of esc are combined together with bytes of e to form an encoding. However without knowing precisely what the bytes are of e, and where the decoding starts, we have no way of knowing how many bytes will be used from esc to form this crossing encoding.

To maintain our ability to reason about syntactic candidates, we define $esc.start$ to be the those bytes that will get be used together with the previous candidate to form an instruction. The precise choice of how many bytes are included is arbitrary since it is fully depended on e and the decoding stream, however in the contexts where we will use them we will make sure to have an exhaustive list esc where $esc.start$ is defined for every possible desired position.
All the other properties are derived from starting a decoding stream after the bytes of $esc.start$
Note that stating that $esc.start = esc$ implies that streams have converged at the start of esc. Although there are other ways of converging as well.

Converging:
If after $esc.start$ we can decode one or more instructions, and the all the bytes in $esc.start$ are consumed, then necessarily we have converging streams. This happens when $esc.start$ contains no bytes or $esc.start$'s ending coincides with an ending of an encoding that makes up the syntactic candidate. $esc.converging$ denotes if this is the case or not. We assume esc is not convergent unless stated otherwise.

Internally consistent:
If after $esc.start$ we have more bytes than the maximum encoding length defined in the ISA. It could happen that decoding directly after $esc.start$ would never yield a valid instruction. This has the consequence that if this esc was used in a chain it would always break the validity of stream. This property is denoted by $esc.internally\_consistent$. By default we assume that esc is internally consistent.

End:
$esc.end$ is similar to $esc.start$ and represents those bytes that will form an instruction together with an following syntactic candidate. We obtain $esc.end$ by putting a decoding stream on the byte after $esc.start$, assuming esc is internally consistent and not converging, we have bytes that can not be decoded on their own and require the addition of bytes from the start of the following syntactic candidate.

In contrast to start, end is actually a list of byte sequences. Due to anything in $\mathcal{N}$ being allowed to be inserted between any two encodings. It is allowed to insert these in between two encodings inside esc as well. This could have as a consequence that the internal decoding stream gets altered and a different set of bytes is exposed at the end. Note that every non-empty entry for end still represents a valid path from start to end.

The precise construction is found in section IX-D.

Is fully embedded:
This property denotes if the syntactic candidate is fully embedded for a preceding candidate. Note that if this property is set to yes, the other properties are ill-defined, and also necessitates that the length of the syntactic candidate is at most the maximum encoding length - 1.

B. Concatenation on extended syntactic candidates:

We will expand the notion of concatenation for extended syntactic candidates such that the validity of an underlying divergent stream is always maintained. We assume this divergent stream starts after the first byte of the start attribute of the first candidate in the sequence. Also implying that concatenation is undefined for sequences starting with fully embedded candidates.

Assume we have a sequence $(esc_1, ..., esc_n)$ and we want to append esc_{new}. We have two options based on if esc_{new} is fully embedded:

If esc_{new} is not fully embedded we have two cases depending if esc_n is fully embedded or not. If esc_n is not fully embedded then concatenation is only allowed if $esc_n.end$ together with $esc_{new}.start$ create an instruction. Meaning that the last encoding of esc_n and the first of esc_{new} cross each other.

If esc_n is fully embedded, then instead of looking if $esc_n.end$ forms an instruction together with $esc_{new}.start$, we look if it holds for the sub-sequences, starting from the last non-fully embedded encoding e, up to esc_n. i.e we test for if $e_i, ..., esc_n, esc_{new}.start$ forms an instruction.
For both cases the original divergent stream converges based on if esc_{new} is converging or not.

If esc_{new} is fully embedded, we will just append it as long as it doesn't make the sub sequence starting from the last non-fully embedded instruction esc_i up to esc_{new} longer than the ISA limit of 15. The stream converges

based on if the sequence from $esc_i.end$ to esc_{new} forms an instruction.

C. Graph reduction

To start testing, we will create a set graph $G = (V, E)$ which represents all possible ways to concatenate syntactic candidates such that all paths maintain the validity of divergent streams. To do so, we will first convert the set of syntactic candidates SC into a set of extended syntactic candidates ESC in the following manner:

1) For each $sc \in SC$, we replace sc with up to 15 different esc in SC, the i'th created esc, we have that $esc.start$ is the first i bytes from sc. The exact amount of created esc is equal to the length of sc if it is below 15. Additionally, we also will create one extra esc that is marked as fully embedded.
2) We filter out all non-internally consistent esc
3) We filter out all fully embedded esc belonging to the first semantic unit
4) We filter out all converging esc belonging to all but the last semantic unit
5) We filter out all non-converging esc belonging to the last semantic unit

Even though we filtered for all converging esc, we still have to separately check that concatenating fully embedded candidates does cause the stream to converge.

Building the graph:
Assume we have n semantic units. First we will create n columns where the column represents the grouping of candidates for a semantic unit.
We will go through the sequence of semantic units from start to end. For the considered semantic unit su_i, we have will consider for every esc which esc_{next} from the succeeding semantic unit can be concatenated as described in IX-B.

If both esc and esc_{next} are not fully embedded and can be concatenated we will create a vertex for both if they do not exist and placing them in the appropriate column. We also will create an edge between them. If a candidates already has a vertex created for them, we will create an edge to the pre-existing vertex instead.

If esc_{next} is fully embedded, we will always create a new vertex. This ensures that the fully embedded vertices always have precisely one incoming edge (reading from left to right) preserving the information of which non fully embedded candidate they belong to.

Properties of the graph
Observe that this graph has 3 important properties:

1) If we have a path in the graph from the first to the last column, this path represents an fully overlapping binary (soundness)
2) If it is possible to create an overlapping binary while only using the given set of syntactic candidates, it will represented as a path (completeness)
3) If no syntactic candidate from a semantic unit su_i has a vertex, then su_i or it's direct adjacent neighbour semantic units needs to change to be enable overlapping. This is assuming the syntactic candidate list is exhaustive
4) Like-wise with the previous point, all semantic units which have syntactic candidates such that there exists an edge to both adjacent semantic units, but have no path between the adjacent semantic units other also needs to change, or its adjacent units need to change to enable overlapping.

D. Dealing with null candidates:

Recall that any syntactic candidate in $\mathcal{N}$ by definition has the property that it has no side-effect onto the input program. This means that they can be placed between any two *encodings*. As a consequence, we can place them also anywhere between two encodings in any syntactic and extended syntactic candidate. For an esc this has the consequence that the end property changes from a single value to a set of possible values.
On a macro level there are two places in which a null candidate can be placed. Inside an esc or between two $escs$. Notice that having an null candidate be inserted between two esc is equivalent to stating inserted null candidate was part of the end of the first esc. Hence we are required to only show that we covered every case of inserting null candidates inside an esc between every two encoding. With the exception of the first esc needed to be handled manually with an extra check.

To cover every case of an null candidate being inserted between two encodings in an esc, we will apply an approach similar to the graph reduction described in IX-C. Recall that the input to the graph reduction is a list of semantic units accompanied with syntactic candidates.
We can convert any esc, which is a sequence of encodings, to a list of semantics and syntactic candidates by taking the semantic of every encoding individually, as well as taking the encoding as only syntactic candidate. With every candidate being one encoding long, we prevent our argument from being completely circular.
This graph reduction changes in three ways.

The first semantic unit now only has a single extended syntactic candidate with the start equaling the start of the original esc.

Secondly, between every column, and after the last column, we add an optional column which are populated by every element in $\mathcal{N}$. In contrast to the normal columns which needs to follow a strict order, these might be skipped or taken.
The way how these optional columns are populated is by having every $n \in \mathcal{N}$ converted (ideally this is already done

during bootstrapping) into an extended syntactic candidate. We assume here that any internally consistent null candidate that can be created by concatenating together multiple null candidates already exists as a separate candidate to allowing us to model every candidate for any n to have a single $start$ and end.
Meaning that every n has at most two extended candidates in these optional columns, one having a single $start$ and end, and if it is smaller than 15 bytes we also have an extra candidate being fully embedded.
We do not include an extra column before the first normal column due to the extra cases we would gain being already handled by the extra end values of previous candidates, meaning we do not have to complicate our model more by changing start to potentially large list.

Finally, we change the filter that last candidates should be converging, but to instead being divergent. With all bytes that are not able to be decoded being put as an entry in the $esc.end$ list.

Concatenation would still take place as described earlier, but instead of checking for $esc.end$ we check for if any $esc.end$ has the desired property. In an implementation of the system we would also probably keep track of which null candidates should be included where to obtain the specific end value.

X. Encoding selector

If the graph reduction produces a graph which has a path from the first to the last semantic unit, we know there is at least one way of generating of an overlapping binary. The encoding selector is responsible for finding a suitable or preferred path and emit all of the encodings which are associated with that to an output file, marking the end of the system.

One key assumption thus was left implicit, namely that an encoding has a constant value. However this assumption typically does not hold modern execution environments. The location of which pieces of data are loaded from, or the precise offsets of jump instructions are often determined at link time.
This could potentially break any overlapping of our binaries by changing the values of an encoding into something that would invalidate another decoding stream.

To counter act this we could incorporate two solutions. Assume we have some encoding e which has bytes $(b_i, ..., b_j)$ that are changed during link time.
First we could try overlap binaries in such a way that in the other decoding stream would accept arbitrary values for $(b_i, ..., b_j)$. This would typically happen in the case that in the other decoding stream it would see $(b_i, ..., b_j)$ as operand values.
Secondly it might be possible to pre-determine the precise offset depending on the executable/object format in question. If we happen to have multiple ways to overlap an binary, we could look at the length of encodings in each option to see if there is a way on how we can match the possible candidate. The precise methods of how this can be accomplished is left as future work.

XI. Multi semantic analyzer

If our graph reduction shows that our current set of candidates are unable to form an overlapping program. We feed our input program and our graph reduction into the "multi-semantic analyzer". In contrast to the Single semantic analyzer at the beginning of the system, this is allowed to change the apply diversification that impacts multiple semantic units at a time. Allowing for a much greater diversity in techniques.
We first will determine a set of semantic units $\mathcal{C}$ which we require to be changed before feeding the output of the multi semantic is recursively inputted into our system from the start. This can be done after all of the items in $\mathcal{C}$ have changed, or just one allowing for quicker testing whether the newly changed item overlaps or needs to be changed again.
We consider two main sets for the construction of $\mathcal{C}$. Both of these places a different emphasizes on only changing semantic units which we know requires need to be changed, hoping that with each iteration we can smaller this set until they are all empty and hence the binary is overlapping.

The first set is the set of semantic units which have edges incoming from adjacent semantic units, but no path exists from one adjacent semantic unit to the other.
The second set is the list of semantic units su_i to su_j such that su_i has an edge with su_{i-1}, and su_j with that of su_{j+1}, but no semantic unit in between has an edge to any other semantic unit from the input program. In the best case only every other element in this set needs to be changed due to every change being able to create a path between two adjacent semantic units.
We speculate that the actual transformations could be build on top of existing compiler infrastructure, where for a small code unit it would go through the myriad of program transformation passes in different ways.

XII. Claim of near completeness

We present two arguments as to why, in-case of no results from our system on a couple of test-cases, that it would be unlikely for the vast majority of other test cases as well. Note that many of these arguments are based completely on conjecture and require further work to be proved/disproven.

A. No single transformation could fix up overlapping with crossing encodings

For overlapping binaries which that contain only full embedded encodings, we typically have a structure in which for one decoding stream, we have large instructions with arbitrary arguments, and with target code being placed into these arguments.

We argue that this does not hold up for overlapping binaries with crossing encodings. This due to the fact that since one encoding must be crossing, we have that a semantic piece that needs to encoded to maintain the semantics of an input program necessarily alters part of the start of the byte sequence of an instruction in the other stream. This part is also responsible for determining the length of the instruction, meaning that if we alter the input's program semantics, the length of this other function could also change. Making overlap less likely.
This breaks with the notion that we can create a fixed structure, which a single transformation would need to map into, akin to what we have for fully embedded programs.

We also would argue that due to the large amount of diversity added during the first stage of our system, we have enough diversity on a byte level such that the effects of each semantic transformation would be included in a diversification of another semantic transformation.

B. completeness of semantic transformations

The following argument will counter the notion of "adding more transformations should change our results", the idea behind this is that if we included a large amount of program diversifications of all available program diversifications, there would only be 2 scenarios in which we adding more diverfisications could change our results.

The first being that from the current day, there are a orders of magnitude more program transformations discovered which are not covered before by a permutation of existing transformations. We consider this unlikely.
META NOTE: I tried looking for a SoK related to all known program optimizations and could not find one. Which makes me a bit uneasy with this claim because I would ideally like to point that the growth curve of the amount of all known optimizations/diversifications are geting stale :(

The second would scenario would include that we would not need many extra transformations, but that a couple of extra would suffice. However if this is the case that means that the growth function of the ratio between overlapping added per transformation would be rather high. If this were the case that the probability per direct result of a high growth function would be that we most likely be more in the "we are really far away and require many more transformations" than the "we are really close and need just a few".